Empirical Software Engineering (2024) 29:23
https://doi.org/10.1007/s10664-023-10368-x

Language usage analysis for EMF metamodels on GitHub

Önder Babur[1,2] · Eleni Constantinou[2,3] · Alexander Serebrenik[2]

Accepted: 13 July 2023

Abstract

Context EMF metamodels lie at the heart of model-based approaches for a variety of tasks, notably for defining the abstract syntax of modeling languages. The language design of EMF metamodels itself is part of a design process, where the needs of its specific range of users should be satisfied. Studying how people actually use the language in the wild would enable empirical feedback for improving the design of the EMF metamodeling language.
Objective Our goal is to study the language usage of EMF metamodels in public engineered projects on GitHub. We aim to reveal information about the usage of specific language constructs, whether they match the language design. Based on our findings, we plan to suggest improvements in the EMF metamodelling language.
Method We adopt a sample study research strategy and collect data from the EMF metamodels on GitHub. After a series of preprocessing steps including filtering out non-engineered projects and deduplication, we employ an analytics workflow on top of a graph database to formulate generalizing statements about the artifacts under study. Based on the results, we also give actionable suggestions for the EMF metamodeling language design.
Results We have conducted various analyses on metaclass, attribute, feature/relationship usage as well as specific parts of the language: annotations and generics. Our findings reveal that the most used metaclasses are not the main building blocks of the language, but rather auxiliary ones. Some of the metaclasses, metaclass features and relations are almost never used. There are a few attributes which are almost exclusively used with a single value or illegal values. Some of the language features such as special forms of generics are very rarely used. Based on our findings, we provide suggestions to improve the EMF language, e.g. removing a language element, restricting its values or refining the metaclass hierarchy.
Conclusions In this paper, we present an extensive empirical study into the language usage of EMF metamodels on GitHub. We believe this study fills a gap in the literature of model analytics and will hopefully help future improvement of the EMF metamodeling language.

Keywords Model-driven engineering · Metamodeling · EMF · Mining software repositories · Model analytics · Software linguistics

Communicated by: Alexandre Bergel

✉ Önder Babur
onder.babur@wur.nl

Extended author information available on the last page of the article

1 Introduction

Modeling and model-* (e.g. model-driven, model-based) techniques are widely used approaches to tackle the increasing size and complexity of software-intensive systems (Brambilla et al. 2017; Tekinerdogan et al. 2019), e.g., in cyber-physical and automotive systems, models are central components of the corresponding software (Mohamed et al. 2020; Broy et al. 2012). Tool support (e.g. the Eclipse Modeling Project[1]) and emerging language workbenches (Erdweg et al. 2013) further pave the way to the success of those approaches. Various communities have put forward different modeling languages (MLs), ranging from more general ones, such as Unified Modeling Language, to domain-specific languages (DSLs). MLs in general, but DSLs to even a greater degree, address a relatively narrow category of users, since these languages are domain-specific, represent (possibly in-house) technology niches (Herrmannsdoerfer et al. 2010) or dedicated groups (Hebig et al. 2016) (e.g. proponents of the model-* paradigm). This is in contrast with other types of languages, e.g. general-purpose programming languages (GPLs) and further with typical natural languages (NLs), which appeal to a much wider variety and number of users. A large user base for GPLs, along with the developments in digitalization and open source software, has led to an abundance of data (i.e. instances of the language, programs) and hence made it possible for researchers to apply techniques from software linguistics (Favre et al. 2010), source code analytics and mining software repositories targeting GPL source code (as the most often analyzed artifact type in repository mining research (de F. Farias et al. 2016)). Favre et al. emphasized the necessity for performing empirical studies on software languages in general, as *“Software Languages are Language Too!"* (Favre et al. 2010). Empirical studies on model-* artifacts, as a particular subset of software languages, have received relatively less attention in research (Tekinerdogan et al. 2019) until very recently. One of the highlights for this line of research would involve mining UML models from GitHub (Hebig et al. 2016), and studying the practices and perceptions of their usage (Ho-Quang et al. 2017).

Metamodels, as one of the main types of model-* artifacts (Combemale et al. 2016), are at the heart of those approaches and software language engineering, as they describe (and restrict) the common elements and their relations in MLs and DSLs. Metamodels are *models* themselves; they are actually the instances of a language one meta-level higher (described by a meta-metamodel) and together they comprise *the meta-language of language definitions*. Studying the language usage of metamodels *en masse* is a key line of research for software language engineering (Herrmannsdoerfer et al. 2010). This is mainly related to the fact that the engineering of a language is itself actually a design process (and in fact a difficult one (Clark et al. 2015)), with two major problem-solving cycles: design and evaluation (Wieringa 2014). Language engineers attempt to capture a focused domain of application with certain expectations (i.e. design cycle), targeting the needs of a very specific category of users (Paige et al. 2000). According to Paige et al., the extent to which those expectations correspond to the user needs, and eventually the needs get fulfilled, is a major factor in the success of the language usability, adoption and longevity (Paige et al. 2000). A proper assessment of the language usage (i.e. empirical cycle) for MLs —both general purpose and domain-specific— is usually overlooked in the literature, e.g., as observed by Gabriel et al. for DSLs (Gabriel et al. 2010). The results of such evaluation provide feedback to the language designer for evolving the language (Tairas and Cabot 2015), e.g. by adding missing constructs and removing unnecessary ones; or for taming the language usage by imposing additional constraints (Favre et al. 2010). This feedback can be expected to be very valuable to the language designer (in a

[1] https://www.eclipse.org/modeling/

simple manner or iteratively as a part of a participatory design framework (Muller and Kuhn 1993)), who otherwise is limited to the traditional means for feedback such as bug reports and questions on forums.

The Eclipse Modeling Project[2] is one of the most popular (Izquierdo et al. 2017) language workbenches and model-* ecosystems, which is based on Eclipse and supports a wide range of modeling standards, languages, technologies and tools (editors, code generators and many more). At its core stands the Eclipse Modeling Framework (EMF) (Steinberg et al. 2008), with a well-established meta-metamodel, called *Ecore*, for describing the abstract syntax of metamodels. One of the key factors in the adoption of EMF/Ecore for a wide range of model-* approaches (as reported in Steinberg et al. (2008)) is its being the de facto reference implementation of the Meta-Object Facility (MOF[3]). MOF in turn is an international industry standard for metamodeling maintained by the Object Management Group.[4]

While EMF metamodels are reported to be widely used (Babur et al. 2017; Kolovos et al. 2015), there has been limited effort from the community to study their actual usage in the wild with a particular language-level focus. With the exception of the study by Herrmannsdoerfer et al. (2010), the few existing studies take a holistic approach to consider the whole technology model of EMF (Heinz et al. 2020) including other types of artifacts in model transformation and code generation (Di Rocco et al. 2020) — see Section 10 for a more detailed discussion on related work. Studying the usage of the EMF metamodeling language could result in useful input for the next design iteration of the EMF languages and ecosystem, as well as the MOF standard itself.

In this paper, we perform an extensive usage analysis for EMF (Ecore) metamodels on GitHub. Our overarching research question is *“How are EMF metamodels used in the wild?"*. More specifically, our goal is to analyze EMF metamodels for the purpose of investigating their usage with respect to the language features, from the point of view of ML/DSL designers in the context of open source software in the wild. Note that we limit our scope to open source. However, EMF is also used in industry (Steinberg et al. 2008; Hutchinson et al. 2011) and a study of EMF metamodels in industry would possibly reveal a different picture — see Section 10 for further discussion of related work on EMF in industry. Following this, we formulate fine-grained research questions on each particular aspect of language usage in Section 8, also shortly outlined in Table 1. We aim to study the language usage in engineered (as opposed to student or toy) projects on GitHub, therefore we discuss the collection of engineered project data as well as the generic architecture of our approach to be used in future model analytics studies.

The contributions we make in this paper are as follows:

1. A generic workflow for performing empirical analyses on model-* artifacts using our state-of-the-art model analytics framework SAMOS (introduced in Section 2.3),
2. A curated dataset of EMF metamodelsavailable as GitHub URLs from engineered projects as a basis for empirical analyses,
3. An extensive study on language usage for EMF metamodels on GitHub, with a particular focus on syntax. Our primary findings are as follows:
   - The most used metaclasses are not the main building blocks of the language such as *EClass*, but rather auxiliary ones such as *EStringToStringMapEntry*. Some of the

[2] https://www.eclipse.org/modeling/

[3] https://www.omg.org/mof/

[4] https://www.omg.org/

Table 1 Outline of research questions, sub-research questions and their corresponding sections

Research Question	Sub-research Question
1 - Metaclass Usage (Section 8.1)	1.1 - What metaclasses are the most used?
	1.2 - What metaclasses are rarely used or not used at all?
2 - Feature/Relationship Usage (Section 8.2)	2.1 - What features/relationships are not used, or rarely used?
	2.2 - What features/relationships are not used to their full multiplicity?
3 - Attribute Value Usage (Section 8.3)	3.1 - What attribute values are not used, rarely used or used outside their theoretical range?
	3.2 - Which attributes (already having default values) do not have the most used value as default value?
	3.3 - Which attributes have most often used values?
4 - Annotation Usage (Section 8.4.1)	4.1 - What are the often used types of EAnnotations, as indicated by their source attribute?
	4.2 - What EStringToStringMapEntry keys are frequently used with each other and the corresponding EAnnotation types?
	4.3 - For those frequently co-occurring sets of keys, are there also frequently used values?
5 - Generics Usage (Section 8.4.2)	5.1 - What percentages of EClassifier and EOperation instances contain type parameters?
	5.2 - Which specific types of generics are used overall?
	5.3 - How often do EClass instances use generic supertypes and EOperation instances generic exceptions?

metaclasses, namely *EFactory* and *EObject*, are not at all used in the EMF metamodels.
- Several metaclass features and relations, e.g. *eKeys*, are almost never used.
- There are a few attributes which are almost exclusively used with a single value (e.g. *EEnum.serializable* almost always true) or illegal values (e.g. multiplicities set to large negative numbers).
- Some of the language features such as special forms of generics are very rarely used.
- Based on our findings, we provide suggestions to improve the EMF language, e.g. removing a language element, restricting its values or refining the metaclass hierarchy.

The remainder of the paper is organized as follows. In Section 2 we review background on software and model analytics, model-* approaches, DSLs, metamodeling and finally SAMOS model analytics framework (Babur et al. 2022) as the tool supporting the analyses in this paper. We give an overview of our approach and the generic workflow. The data collection and filtering phases are elaborated in Sections 4 and 5. In Section 6 we describe how we extract, represent and query information from the metamodels as well as the repository metadata. Following Section 7 on some key statistics on our final dataset, we provide detailed empirical analyses on various aspects of language usage (Section 8): metaclass usage, feature/relationship usage, attribute value usage, and particular usage of specific parts of the language (i.e. annotations and generics). We then discuss the threats to validity (Section 9), related work (Section 10) and conclude with several pointers to future work (Section 11).

2 Background

In this section, we review background concepts related to our study.

2.1 Corpus and Software Linguistics, Model Analytics

Corpus linguistics is a branch of linguistics, the study of language, with distinguishing characteristics such as (a) empirical nature, (b) focus on actual patterns of use in natural texts and (c) utilizing large collection of natural texts via computational analysis (Biber et al. 1998). It aims to uncover how the language users actually exploit the resources of their language, rather than e.g. performing theoretical analyses on the language. While natural languages are the original and commonplace domain for corpus linguistics, in recent years there has been considerable progress in that direction for software languages, typically for general-purpose programming languages (Favre et al. 2010), but also domain-specific languages (Lämmel and Pek 2013). For Java, examples would include general language usage studies (Qiu et al. 2017; Grechanik et al. 2010) as well as more specific ones, e.g. focusing on cycles among classes (Melton and Tempero 2007).

MLs —both general purpose ML such as Unified Modeling Language (UML) or domain-specific ones—, on the other hand, are a subset of software languages typically targeting a smaller set of users compared to the GPLs. This implies a relatively low volume of data to build corpora and perform corpus analyses. Nevertheless, we still need such analyses (Tekinerdogan et al. 2019; Babur 2019) for MLs, as they need to match the very specific expectations of the target users and possess qualities such as simplicity and brevity for successful adoption (Paige et al. 2000). There is a recent interest in building corpora of model-* artifacts (typically from GitHub) and performing various empirical analyses on them, including language usage analysis. We discuss such studies further as related work in Section 10.

2.2 Model-* Approaches, Domain Specific Languages and Metamodeling

Modeling and model-* approaches, such as model-driven software engineering and model-based systems engineering, are among the major paradigms introduced for tackling with the ever-increasing complexity of developing and managing software-intensive systems (Andova et al. 2012). Those approaches rely on a few building blocks (Combemale et al. 2016): modeling as an activity for capturing domain knowledge in an abstract and convenient manner, analysis and/or verification tasks to reason about the systems and assure certain qualities, and finally generative techniques such as model transformation and code generation to realize the systems. The modeling activity is driven by a modeling language which characterizes the set of possible models (Combemale et al. 2016). There are numerous modeling languages, ranging from general-purpose ones such as UML to domain-specific ones, custom-tailored for different purposes.

Metamodels, in the model-* technical space, are the main artifacts for specifying the abstract syntax of a modeling language, i.e. the essential building blocks without concerns about the concrete representation. Following the introduction of UML, Object Management Group pushed forward the metamodeling standard MOF (Information technology 2005). Essential Meta-Object Facility (EMOF) is in turn a lightweight core subset of MOF, intended to match object-oriented language structures as well as serialization capabilities. Simply put, EMOF suggests a four layered linguistic architecture: **M3** where the MOF meta-metamodel resides, **M2** with metamodels for describing (domain-specific or general purpose) modeling languages, **M1** for the individual models and **M0** for the model instantiations (e.g. in the form of objects).

MOF being a standard itself, the de facto industry implementation of MOF is Ecore in Eclipse Modeling Framework (EMF) (Steinberg et al. 2008). While in recent years several new language workbenches have been introduced, the EMF ecosystem remains to be one of the most popular platforms for model-* approaches and domain-specific languages (Izquierdo et al. 2017). As getting a grasp of Ecore is essential for understanding the studies in this paper, below we give an overview of its major components. For full details, readers are referred to the EMF reference book (Steinberg et al. 2008). Figure 1 depicts the core part of Ecore. Note that Ecore deliberately prefixes all elements with "E" to avoid confusion with their counterparts in MOF/UML or Java.

We start by elaborating the *metaclasses*, i.e. the set of classes, in Ecore. Ecore allows collecting domain concepts in *EPackages*. Such *EPackages* can contain *EClasses* representing domain concepts, with *EAttributes* as properties and *EReferences* as (one end of) associations with other concepts, as well as *EOperations* as service signatures (i.e. not implementations) with possible *EParameters* and *EException* declarations. Certain elements can have types; e.g. *EAttributes* can be typed by a built-in primitive type such as *EInteger*, a custom *EDataType* or *EEnum*. There is also support for inheritance among *EClasses*. Besides the main concepts and relationships, metaclasses contain further attributes denoting whether an *EClass* is abstract, an *EReference* implies containment (to distinguish composition and aggregation in UML terms), multiplicities for elements/associations, and so on. *EAnnotations* are a flexible part of Ecore, serving different purposes ranging from documentation to model transformation metadata and language extension. EAnnotation contains an important feature, named *details*, which enables the use of map-typed features of the class *EStringToStringMapEntry*. This class represent key-value pairs and are supported through a special behaviour (e.g. by the code generator), allowing the use of maps.

Another metaclass to mention is *EFactory*, which allows creating instances of model elements. It is used, for instance, in the Ecore API for programmatically building models and in code generation. *EObject* is the implicit common superclass of all Ecore classes. It can be used, again through the programmatic API, for dynamically creating objects; however this is achieved through its subclasses such as *DynamicEObjectImpl* and not by directly instantiating *EObject*. *EFactory* and *EObject are associated with the runtime operation of EMF rather than the metamodeling itself.*

In Ecore, a distinction is made with respect to the features and relationships. Following Fig. 1, there are two types of features/relationships: non-derived (shown with black arrows) and derived ones (shown with blue arrows).[5] As an example, the non-derived feature *eStructuralFeatures* contains all the structural elements in an *EClass* (which can contain both *EReference* and *EAttribute*). The derived relationships of *EReferences* and *EAttributes*, are merely derived from *eStructuralFeatures* and are subsets of it containing correspondingly *EReference* and *EAttribute* instances only.

A particularly interesting feature of Ecore is the support for generics, as shown in Fig. 2.[6] Generics were introduced into Ecore in version 2.3 to better support the corresponding constructs in Java. The main concept is *EGenericType*, along with a handful of relationships. With a combination of those relationships, Ecore can represent non-generic types (see paragraph

[5] Taken from https://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/package-summary.html, accessed on 05.12.2020.

[6] Taken from https://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/package-summary.html, accessed on 05.12.2020.

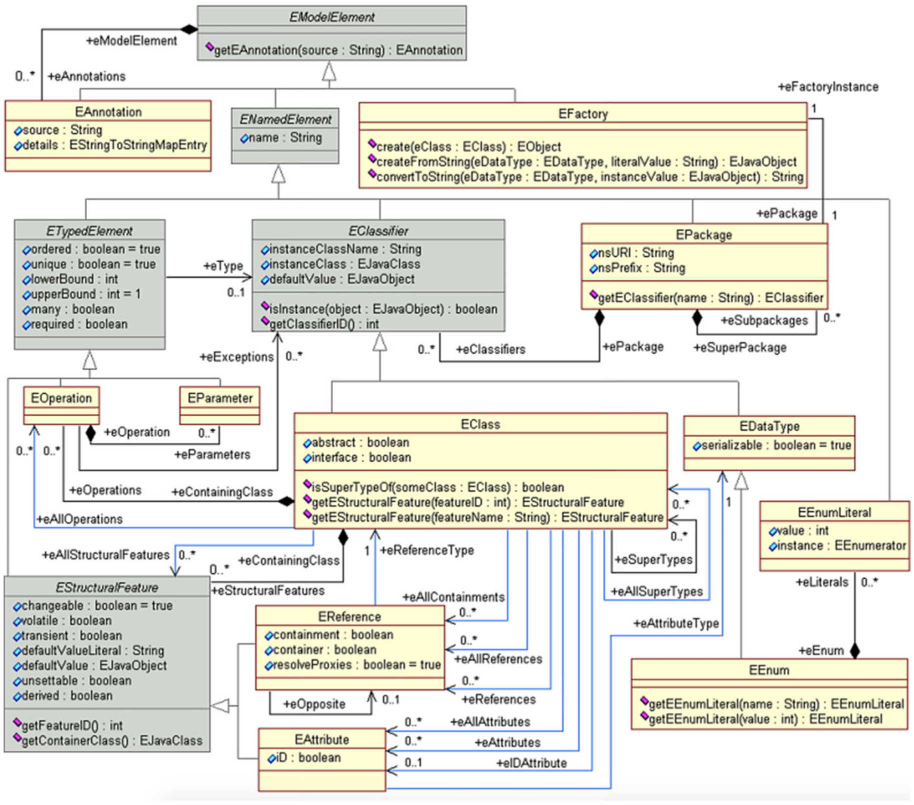


Fig. 1 Ecore core components along with their attributes and relationships

below), simple generic types specified by *eTypeParameters* (e.g. A) coming from an *EClassifier*, upper- or lower-bounded types (e.g. `A extends B` or `A super B`), parametrized types (e.g. `A<B>`) and wildcard types (`?`).

Non-generic types are also internally represented via *EGenericTypes*, though with an appropriate composition (e.g. no bounds and no type arguments) so that the element effectively has a non-generic type. While it is not visible and of immediate significance to the language user, this information will have implications and be taken into consideration in this study, and will be discussed later in Section 8.

2.3 SAMOS Model Analytics Framework

SAMOS (Statistical Analysis of MOdelS) is a state-of-the-art framework for large-scale analysis of models (Babur 2019; Babur et al. 2022). The main approach is to treat models as data (similar to documents in the Information Retrieval technical space) and apply document clustering to models.

Figure 3 illustrates the workflow of SAMOS, with key steps of the workflow and several application areas. The workflow starts with a feature extraction based on the metamodel. Features can be, for instance, simple names of model elements (conceptually similar to the vocabulary in documents) or larger fragments of the underlying graph structure such as n-

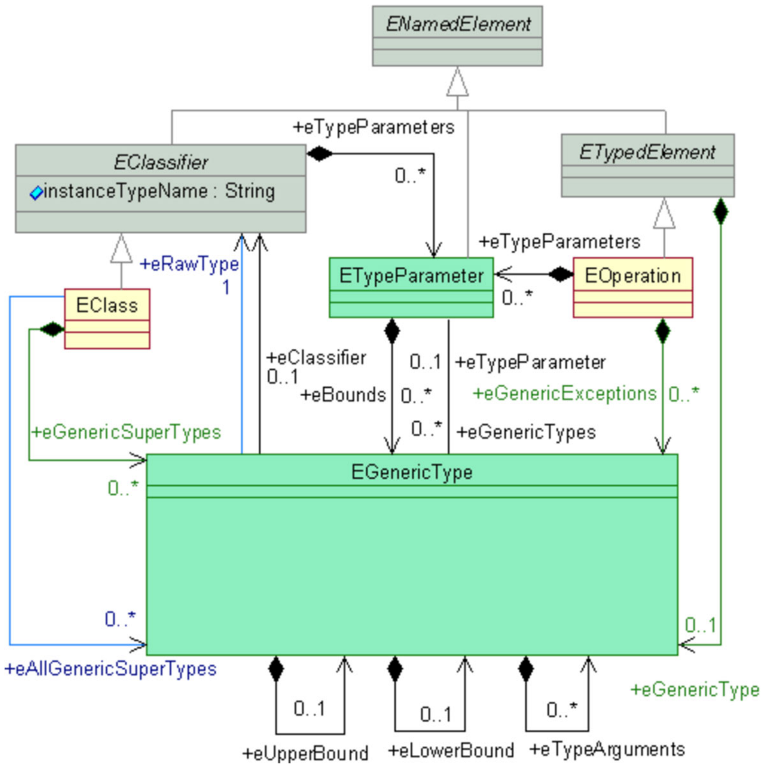


Fig. 2 Part of Ecore associated with generic types

grams or subtrees. To exemplify, see the graph in Fig. 4 for a simplified representation of an EMF metamodel with types and names of model elements, as well as the relationships. SAMOS can extract, for instance, unigrams from this metamodel as follows (see Babur et al. (2019) for the full example):

- $v_0 = \{name : BIBTEX,\ \ type : EPackage\}$,
- $v_1 = \{name : LocatedElement,\ \ type : EClass,\ \ abstract : true\}$,
- $v_2 = \{name : location,\ \ type : EAttribute,\ \ eType : EString,\ \ lowerBound : 1\}$,
- ...

as well as the following bigrams:

- $b_1 = (v_0,\ \ contains,\ \ v_1)$,
- $b_2 = (v_1,\ \ contains,\ \ v_2)$,
- ....

A major step in the SAMOS' workflow is computing a term-frequency based Vector Space Model (VSM)(Manning et al. 2008), using various schemes. These schemes range from matching schemes (e.g. whether to match metaclasses or not), weighting schemes (*EClass* features having higher weight than *EAttribute*) to NLP techniques such as stemming and synonym checking. Applying various distance measures on top of the computed VSM, suitable to the problem at hand, SAMOS applies different clustering algorithms and can output automatically derived cluster labels or diagrams for visualization and manual inspection and

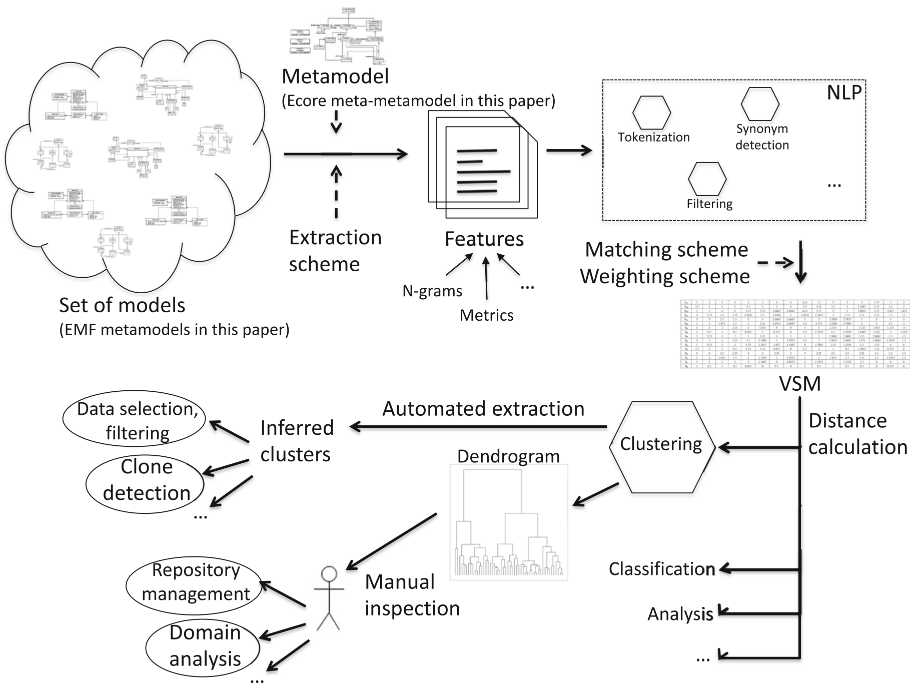


Fig. 3 Overview of the SAMOS workflow

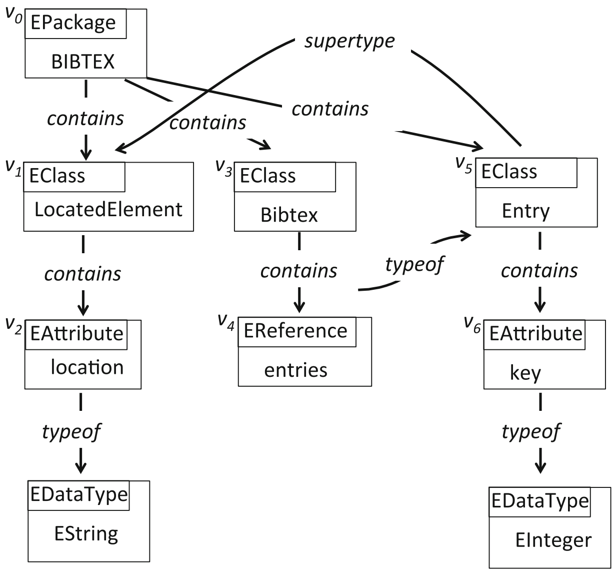


Fig. 4 A simplified graph representation for (part of) an EMF metamodel (Babur et al. 2019)

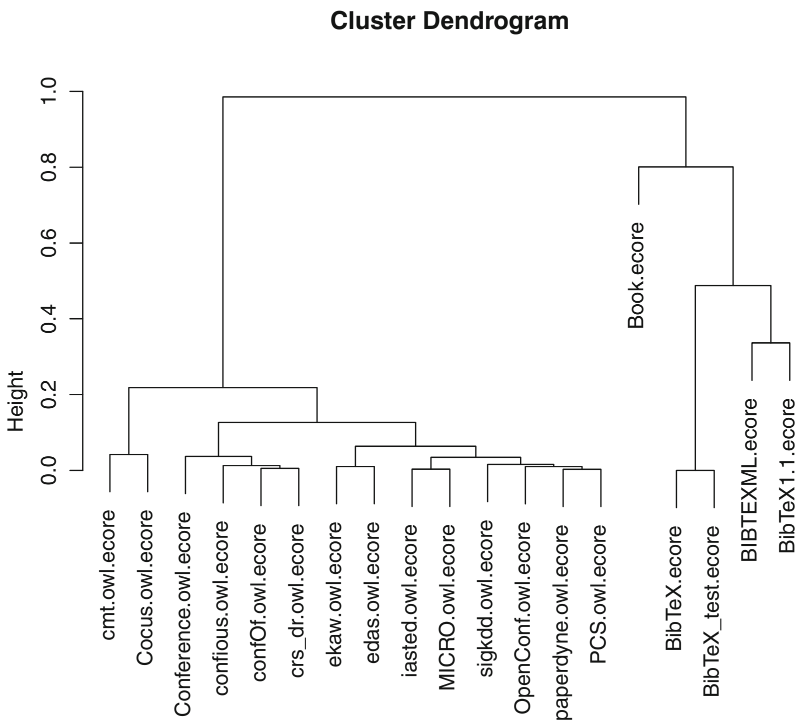


Fig. 5 Plot of the dendrogram as output of domain clustering (Babur et al. 2022)

exploration. An example dendrogram from an illustrative use case (Babur et al. 2022) is depicted in Fig. 5, where SAMOS distinguishes two clusters of metamodels corresponding to the domains of conference management and bibliography management.

SAMOS has so far been validated for a variety of model types including EMF metamodels, feature models, business process models, statecharts and industrial domain-specific models; and for different applications such as domain clustering, architectural analysis and clone detection. More details on SAMOS can be found in our previous work (Babur 2019; Babur et al. 2022), while readers can refer to Basciani et al. (2016) for a comparable tool for metamodel clustering.

3 A Generic Workflow for Modeling Language Usage Analysis

In this section we present an overview of our generic workflow and the steps performed in our study. We adopt a sample study research strategy (Stol and Fitzgerald 2018) in this study. We collect data from open source repositories in an unintrusive way, and by analyzing the repository data, we aim for formulating generalizing statements about the artifacts under study. Our workflow consists of four major components: (1) data collection, (2) data filtering, (3) feature extraction and representation in a graph database, and finally (4) querying, analysis and visualization using graph database queries and a statistical computing language. The generic nature of the workflow follows from the fact that it can be instantiated for multiple modeling languages; therefore we consider this workflow and the corresponding toolchain as a major contribution in this paper. We study EMF metamodels in this paper based on the motivations elaborated in Section 1, but this study can be replicated for other types of models supported by SAMOS (e.g. UML or feature models). Certain parts of the workflow (notably the query and analysis scripts) should be adapted correspondingly to the modeling language and the analysis goals.

The overall workflow is depicted in Fig. 6. The data collection part involves gathering data from various sources (GitHub, SWHeritage, Eclipse) using various methods (e.g. GitHub code search API vs. GHTorrent and GitHub REST API), in order to have a dataset as complete as possible. The result is a set of repositories with metamodels, along with metadata such as owner and creation date of each repository. A data filtering phase is intended to remove non-engineered repositories (Munaiah et al. 2017) (e.g. student projects), duplicated repositories and unparseable/unprocessable (by SAMOS) metamodels. We obtain a final dataset of metamodels and their metadata to perform our analysis on, which is then passed to the data extraction (notably bigrams out of the metamodels) phase in SAMOS. The information is then imported into Neo4j. Finally, a combination of Neo4j queries and R scripts are used to analyze the data and report/visualize the results. In the next sections we elaborate each component of our workflow (particularly for EMF metamodels given the subject of this study) in detail.

4 Data Collection

In this section, we describe the data sources and mining methods we have used for obtaining EMF metamodels to perform our analysis. We formulate our sampling strategy along the guidelines by Baltes and Ralph (2020). The theoretical target population of our study is the set of all EMF metamodels in public engineered projects on GitHub. However, the only access we have to the whole population is through querying. Due to the limitations of the querying mechanisms offered by various sources at the time of conducting this study (GitHub itself or GHTorrent) as well as the high resource demand (computation time, query limit per hour), we have adopted additional inclusion criteria: considering only the main/default branches of non-forked projects, and only in their most recent version (i.e. no version history). The decision to study only engineered projects further imposes the exclusion of non-engineered ones (Munaiah et al. 2017). Also, duplicated projects are excluded to avoid overestimation of metamodel prevalence (Lopes et al. 2017) (to be elaborated in Section 5).

4.1 Data Sources

Various data sources can be used as the starting point for our search, as multiple platforms host open source software in general, as well as domain-specific repositories. We have investigated the data sources in the context and scope of this work, with the goal of answering the following questions: (a) on which platforms do EMF metamodels typically reside and (b) given the recent popularity of GitHub in recent Mining Software Repositories (MSR) studies, what is its adequacy/representativeness for the EMF metamodels in open source (cf. many studies claiming that it is representative (Cosentino et al. 2016))? Based on the related work

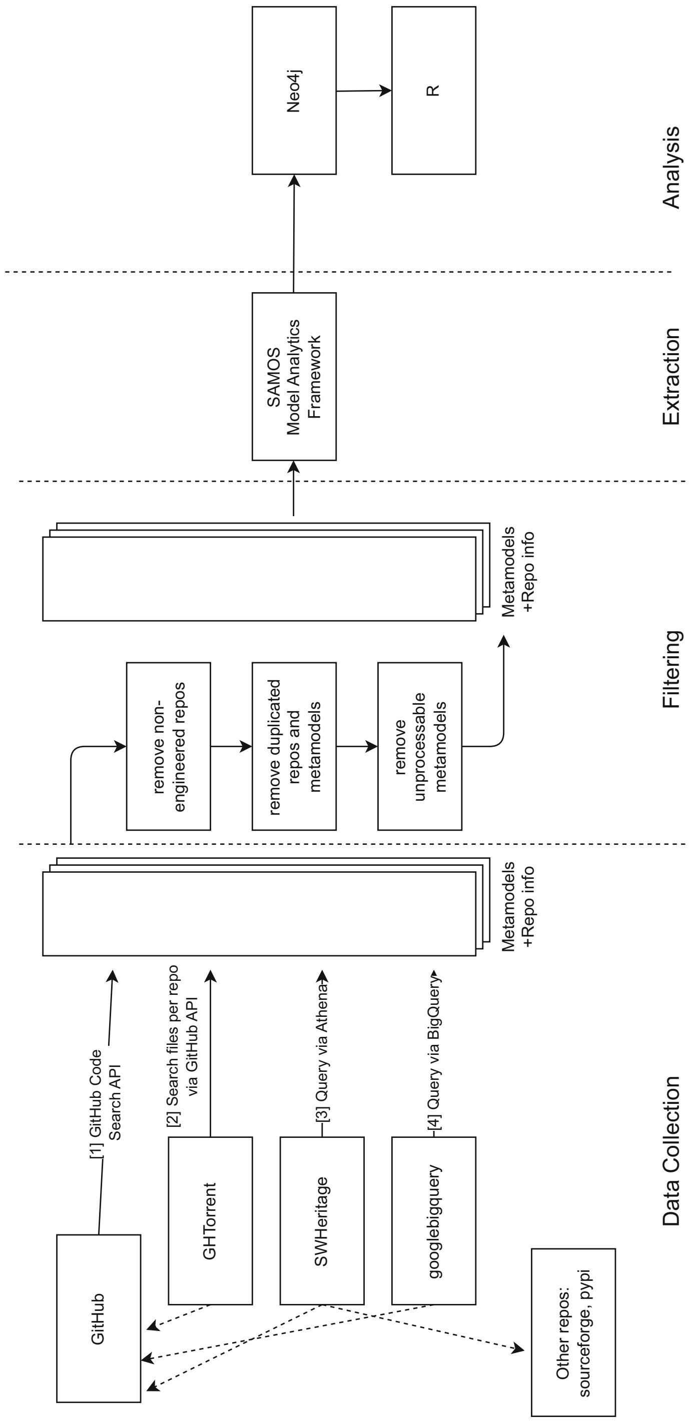

Data Collection
Filtering
Extraction
Analysis
GitHub
GHTorrent
SWHeritage
googlebigquery
Other repos: sourceforge, pypi
[1] GitHub Code Search API
[2] Search files per repo via GitHub API
[3] Query via Athena
[4] Query via BigQuery
Metamodels +Repo info
remove non-engineered repos
remove duplicated repos and metamodels
remove unprocessable metamodels
Metamodels +Repo info
SAMOS Model Analytics Framework
Neo4j
R

Fig. 6 Overview of our generic modeling language usage analysis workflow

(see Section 10), we have identified the following candidate sources for searching for EMF metamodels:

1. GitHub, as popularly used by many MSR researchers (Kalliamvakou et al. 2016),
2. Other popular platforms indexed by the Software Heritage (SWH) dataset (Pietri et al. 2019), which includes, besides GitHub, the following: GitLab, Google Code, PyPi, Debian, Gitorious,
3. Repositories of Eclipse projects, since EMF is part of the Eclipse ecosystem (as used by other researchers (Kögel and Tichy 2018)),
4. Other domain-specific repositories e.g. ATL Metamodel Zoo[7].

In our study we have initially included the first three items, and ignored the last group of repositories as they are typically small, static and volume-wise insignificant compared to other widely used repositories. As an explicit example, ATL Ecore Zoo was populated with ∼300 metamodels in 2010 and has not changed since. GitHub, as opposed to this, consists of millions of projects, hosting thousands of EMF metamodels (see Babur et al. (2017) for a preliminary study). We report the following preliminary observations:

- Our search in the SWH dataset yielded a total of 19,773 repositories (see Section 4.2 for details on the method), with the following decomposition: 18,610 from GitHub, 73 from GitLab, 2 from PyPi, 49 from Gitorious, 1000 from Google Code[8] and 39 from Debian. We conclude that a vast majority of the relevant repositories reside on GitHub.
- We additionally mined the Eclipse repositories to find out if we can find additional repositories hosting EMF metamodels. Eclipse maintains a number of repositories on its own git servers (1.135 listed in the Eclipse website[9]), while also hosting repositories on GitHub (669 repositories under the owner name Eclipse[10]). While most of the GitHub repositories are mirrors of or empty references to the non-GitHub repositories, there are additional repositories as well. Processing the Eclipse repositories not mirrored on GitHub reveals additionally 1,482 (URL-wise unique, where URL is in the form of `owner/repo/pathToEcoreFile`) metamodels from 89 exclusive Eclipse repositories.

Following the above observations, we opted for limiting our study to EMF metamodels on GitHub. From the data sources we have investigated, GitHub hosts an overwhelming majority of the repositories of EMF metamodels we could find in open source. Nevertheless, we leave a more rigorous investigation (e.g. on the diversity (Nagappan et al. 2013) and representativeness (Baltes and Ralph 2020; de Mello et al. 2015) for each platform), for future work, and carefully formulate our target population as EMF metamodels on GitHub throughout the paper.

4.2 Mining Methods

Even if we limit our data source to GitHub only, there exist various means for searching on GitHub. In this section we discuss, compare and eventually merge the results from different mining methodologies (since each have certain shortcomings). Our goal is to answer the

7 https://web.imt-atlantique.fr/x-info/atlanmod/index.php?title=Zoos

8 Google Code repositories have been discontinued, with some portion transferred to GitHub.

9 https://git.eclipse.org/c/, accessed 15 June 2020

10 https://github.com/eclipse, accessed 15 June 2020

following question: *How can we achieve the highest coverage when searching for EMF metamodels on GitHub?*. A high coverage would imply that we get closer to obtaining the whole frame as imposed in our sampling strategy, which would lead to a richer dataset to begin with, as well as more sound and generalizable empirical studies. Before moving to the mining methods, it is important to note that we follow a common practice in related work (e.g. adopted by Hebig et al. (2016)) and target only main/default branches of non-forked repositories, with the most recent commit. This simplification fits this particular study well, as we are not interested e.g. in the evolution of the metamodels.

We have identified several methods employed in the literature for mining GitHub, summarized as follows.

4.2.1 Using GitHub Advanced Search (Method A)

A common way of finding metamodels is by using GitHub advanced search[11] (Babur et al. 2019; Noten et al. 2017; Kolovos et al. 2015; Härtel et al. 2018). The advantages are relatively fast querying and up-to-date data. However, only files smaller than 384 KB and repositories with fewer than 500k files are searchable.[12] The file size limit might potentially pose a threat for empirical analyses. Furthermore, since the indexing is done internally by GitHub, it is hard to argue about sampling strategies e.g. in terms of completeness (when using *whole frame*) and representativeness.

Execution In December 2019, we performed the following query on GitHub advanced search and crawled the results: https://github.com/search?q=NOT+foofoofoo+extension:ecore&type=Code). The choice of a negated string *foofoofoo* is deliberate (Noten et al. 2017), since no EMF metamodel includes that string in its content according to our preliminary checking. Furthermore, we use the strategy of time-slicing to be able to fetch all the results beyond the limitation of GitHub of retrieving 1000 search results at a time.

4.2.2 Using GHTorrent and GitHub REST API (Method B)

Rather than directly querying GitHub one might first use GHTorrent (Gousios and Spinellis 2012) (an offline mirror of GitHub data, for researchers to query and use efficiently) to retrieve a list of projects, for a subset either clone the repository or retrieve the up-to-date file tree from GitHub REST API and search for the files of interest, i.e., metamodels. This approach, i.e. involving GHTorrent and GitHub REST API, has been used not only for models (Hebig et al. 2016; Heinze et al. 2020b) but for repository analysis in general (Lopes et al. 2017; Gharehyazie et al. 2019). The advantage of this approach is the control of target population and sampling strategies. The disadvantages being a very high time cost (of using the GitHub REST API with API limits), and the fact that GHTorrent dataset (even if used just for the repository list) might be outdated, incomplete and partially erroneous (Lopes et al. 2017).

Execution We first retrieved a list of non-deleted, non-forked repositories using an instance of GHTorrent 2019-06-01 MySql dump. Throughout July-August 2019, we scanned the file tree for *all* those repositories, which we accessed through GitHub REST API, looking for metamodel files. For the REST API, we used 20 access tokens donated by colleagues to speed up the process.

[11] https://github.com/search/advanced

[12] https://help.github.com/en/github/searching-for-information-on-github/searching-code

4.2.3 Using the Software Heritage Dataset (Method C)

The second dataset considered above is the Software Heritage dataset (Pietri et al. 2019) for the metamodel files. One of the advantages of querying SWH is the availability of highly efficient and fast querying on big data infrastructure AWS Athena.[13] However, we note that not all GitHub repositories are indexed in this dataset; the coverage is reported by the creators of SWH to increase in further releases of the dataset. Since the GitHub part of SWH has been scraped in the past by bots in a way similar to GHTorrent, it suffers from similar data freshness issues as GHTorrent.

Execution Using the AWS Athena facility on the SWH dataset version 2019-01-28, we queried first the metamodel files, and their repositories (represented as origin URLs in SHW). We filtered out non-GitHub and forked repositories and for the final list, and similarly to Method (B) above, we used GitHub REST API for searching the most recent file tree for the main/default branches per repository. We adopted this process in order to maintain consistency among the methodologies B-D, as well as to ensure data freshness.

4.2.4 Using Google BigQuery GitHub Dataset (Method D)

We use Google BigQuery GitHub dataset[14] to pinpoint the metamodel files. The advantage of this method is the highly efficient and fast querying on the big data infrastructure. However, we note that not all GitHub repositories are indexed in this dataset; e.g. reportedly only noteworthy repositories with a clear public license are included. We are still interested in engineered projects that might be excluded by other methods.

Execution. Using the web interface[15] of Google BigQuery, we ran a query on February 2020 to find the metamodel files and their repositories. For the non-forked repositories, we went on with the searching for the most recent file tree for the master/default branch.

4.2.5 Results and Discussion

In Table 2 we present the results from different methods. The rows correspond to the four methods discussed above, while the columns correspond to the total number of (1) files found, (2) unique owners, (3) unique repositories and (4) unique paths (i.e. eventually representing a file per path) found by the method. By uniqueness, we mean the uniqueness of the paths, i.e. `/owner`, `/owner/repo` and `/owner/repo/fullPathToMetamodel`, with respect to all the other methods.

Based on these numbers, and our experience while mining GitHub, we make the following observations.

- No single method as proposed above offers the maximum possible coverage, i.e. each method is able to find unique items. Rather, mixed data collection methods should be used.

[13] https://aws.amazon.com/athena/

[14] https://cloud.google.com/bigquery/public-data

[15] https://bigquery.cloud.google.com

Table 2 Resulting numbers from different mining methodologies

method	total owners	unique owners**	total repos	unique repos**	total paths	unique paths**
code search [A]	4,431	1,391	6,726	2,238	85,267	48,164
repo scan [B]	3,042	4	4,493	5	38,999	791
swh scan [C]	3,097	134	4,653	307	39,861	5,545
big query [D]	536	2	718	11	4,547	69
two or more methods	3,090		4,590		38,778	
total/merged*	4,621		7,151		93,347	

Two types of measures are reported per method: total, just plain totals found by the method, and unique, meaning the unique items found by each method (and not the others). The total/merged row is marked with an asterisk, to underline the fact that it is not merely a sum of the columns per method. It represents the unique count for each item, e.g. the total number of unique owners found by merging all four methods' results. For the unique count columns marked with double asterisk, the totalmerged row also takes into account the intersection among the methods (i.e. any items found by two or more methods, given in the row above), thus is larger than the sum of the individual column values per method

– Code search (A) nevertheless yields the largest portion of results. Repo scan (B) and big query (D) do not bring too many unique data points (i.e. owners and repositories) when compared to the other methods.
– While there is some time gap between our execution of the methodologies, by manual inspection of a sample result set we observed the difference is mostly due to the inherent limitations of the individual methodologies: limitations of GitHub code search, incomplete representations by GHTorrent, Software Heritage and Google BigQuery.
– Data freshness is definitely an issue; in the sample we analyzed, we observed quite a few repositories either deleted, made private or renamed. Renaming and redirecting is particularly an issue we tackle in the next section when detecting duplication in our data.
– By ignoring forks and non-main branches, we miss *some* metamodels. For the forks, it is especially evident when the original repository is archived, and the development is continued in the fork.
– By only taking the most recent file tree, we also miss *some* metamodels which once existed but disappeared along the project development.

Finally, we combine the data obtained by using different methods and merge it into a single dataset (i.e. the total row in Table 2) for the next step of data filtering.

5 Data Filtering

So far, our dataset contains quite a few data points which can be a threat to validity of conclusions for our empirical study. First, GitHub is a public platform where everybody can push their projects. Researchers have reported that there are a lot of perils due to the quality and characteristics of those projects, which could introduce bias in empirical analyses (Kalliamvakou et al. 2016). Kalliamvakou et al. (2014), for instance, manually sampled 434 repositories from GitHub and found that only 63.4% of them were for software development; the remaining were used for experimental, storage, or academic purposes, or were empty or no longer accessible. Munaiah et al. (2017) further emphasize the existence of non-engineered repositories (e.g. toy or student projects) and the potential bias they might introduce in MSR studies. Furthermore, there can be a lot of duplication (Gharehyazie et al. 2019) in terms of whole repositories (e.g. due to implicit forking) as well as individual metamodels (e.g. due to repetition from textbook examples), which could further bias the analysis results. We address these two issues of non-engineered repositories and duplication, and the corresponding filtering, in this section. Furthermore, we do a final filtering pass on the metamodel level to eliminate unparseable or unprocessable metamodels.

5.1 Filtering Out Non-engineered Projects

It has been long observed in the literature that public repositories such as GitHub host millions of projects, and provide a very convenient platform for researchers who wish to study software engineering practices in the wild (Kalliamvakou et al. 2016). However, this comes with a very important problem of distinguishing engineered projects from e.g. toy projects or student assignments (Munaiah et al. 2017), since our study in this paper targets the language use of EMF metamodels in engineered projects.

An established tool for classifying GitHub projects as engineered and non-engineered is Reaper (Munaiah et al. 2017). We have observed several disadvantages of Reaper: its dependence on the GHTorrent data (notably the data freshness issues as discussed in Section 4), and relatively slow performance (Pickerill et al. 2020). For these reasons, we opted for PHANTOM-COYOTE[16] (to be referred to as PHANTOM in the rest of the paper), a recent open source tool suite for identifying engineered projects using a number of classifiers directly on the git files and logs (Pickerill et al. 2020). Pickerill et al. report that PHANTOM has a comparable accuracy with Reaper, and does not possess the shortcomings of Reaper as mentioned above.

5.1.1 Classification with PHANTOM

We applied PHANTOM to the metamodel repositories, and for the repositories PHANTOM can find and process (excluding repositories that were deleted or became private), we obtained the verdicts (engineered vs non-engineered) in terms of five different classifiers (merges, commits, committers, integrations, integrators) against two project categories (organization and utility). These two categories of engineered projects are defined by Munaiah et al. (2017) as being similar to the projects contained within repositories owned by popular software engineering organizations such as Amazon, Apache, Microsoft and Mozilla (organization category) and being similar to the projects that have a general-purpose utility to users other (utility category). We then apply the following aggregation: if a repository is labelled as engineered in either project category with respect to majority vote by all the classifiers (as done in other fields for choosing classifiers (Ruta and Gabrys 2005)), we label it for inclusion. We leave it to future work to study the classifiers (e.g. their correlation) and determine an optimal strategy for classifier selection.

5.1.2 Results and Discussion

Table 3 depicts the results per method, as well as the combined dataset.

Based on our results, we make the following observations:

[16] PHANTOM https://github.com/Ionman64/PHANTOM/ and COYOTE https://github.com/joshuaju/COYOTE/

Table 3 Resulting engineered vs non-engineered projects

method	repos processed	engineered	percentage
code search (A) - total	6,637	4,489	68%
code search (A) - unique	2,163	1,290	60%
repo scan (B) - total	4,477	3,198	71%
repo scan (B) - unique	5	4	80%
swh scan (C) - total	4,641	3,342	72%
swh scan (C) - unique	305	252	83%
big query (D) - total	713	571	80%
big query (D) - unique	11	10	91%
total combined	7,064	4,843	69%

We report total as well as unique counts with respect to all the other methods, as we did in Table 2. Note the slight difference in repository counts with Table 2 due to time difference in mining GitHub (see Section 9 for a discussion)

– Percentage-wise, there is no huge difference among methodologies at first glance. However, running the chi-squared test yields a p-value < 0.00001; therefore, there is a statistically significant difference among different methods. Method A contains a larger ratio of non-engineered projects, while D contains the least, possibly thanks to its more strict inclusion criteria (with a clear public license or noteworthy —e.g. excluding empty projects or unchanged forks— are indexed, as reported for Method D[17]).
– The overall percentages of engineered-repositories are on the high end of the range reported by Pickerill et al. (2020). They validated their tool on a large dataset of (1.85 million) GitHub repositories and report mainly the range of 35-40 % for the majority of their classifiers, and ultimately the range 19-96% when all are considered.

We note that the investigation of engineered repositories by Pickerill et al. was applied to (a sample of) the general population of GitHub repositories with typically GPL content. It would be interesting to use Reaper and PHANTOM specifically for metamodel or model-* repositories, and validate its accuracy. We leave this as future work.

We proceed with the combined set of 4,843 repositories containing 72,384 metamodels to the next phase of filtering, i.e. repository deduplication.

5.2 Deduplicating Repositories

Substantial amount of duplication on GitHub, especially across repositories, has been recently observed as a potential threat to studies performed by researchers (Lopes et al. 2017; Allamanis 2019; Spinellis et al. 2020). For the MSR community in general (typically with a source code analysis angle) this has been pointed out e.g. by Lopes et al. (2017), who reported that 70% of the code on GitHub consists of clones. This goes beyond the standard practice of treating forks as near-duplicates thus filtering them out (Munaiah et al. 2017), as duplication exists across repositories without explicit forks (Gharehyazie et al. 2019; Spinellis et al. 2020). In the related work on mining models, however, there are mixed practices on whether to perform deduplication and how to implement it. Hebig et al. (2016) have not performed any deduplication pre-analysis for the UML models, while Heinze et al. (2020b) have done a

[17] https://cloud.google.com/bigquery/public-data

complete file-level deduplication with the goal of building a benchmarking corpus of BPMN models. We have observed high file-level duplication (66% of the dataset being duplicates) in the metamodels on GitHub in our previous clone detection study (Babur et al. 2019). Note that a fine-grained empirical study on clones as done in our previous study on metamodel clones (Babur et al. 2019), or in other related work on programming languages (Lopes et al. 2017) or domain-specific languages (Lämmel and Pek 2013) is out of scope for this paper, and is left for future work.

As in this study we wish to study the language usage of EMF metamodels on engineered projects, it is essential for us to come up with a rigorous approach for identifying those engineered-projects projects and building a deduplicated high quality dataset not to bias the analysis results (Gharehyazie et al. 2019). We have quantified (a) inter-repository metamodel duplication (on the file level) and (b) repository-level duplication as percentages to guide our filtering.

5.2.1 Inter-Repository Metamodel Duplication

We followed the same approach proposed in our previous work (Babur et al. 2019) for detecting metamodel duplicates. We compared the file hashes of the metamodels in our dataset (`MD5` hash as used by SAMOS), and computed a metamodel duplication percentage, in terms of the percentage of metamodels duplicated elsewhere. Following our observation of whitespace differences in metamodels in certain cases (Babur et al. 2019), we calculated the MD5 hash after normalizing the whitespaces in the metamodel files serialized in XML/XMI format. We found 2864 repositories containing (one or more) duplicated metamodels. We used the list of those repositories for the second stage in our duplicate detection process, where we performed repository-level duplicate detection.

5.2.2 Repository Duplication

Lopes et al. (2017) discuss the non-trivial levels of duplication in the source of GitHub projects, and present a method for detecting file-level duplication across GitHub projects. We could not directly use their method in our work due to the heterogeneous nature of the repository languages we had (outside the specific set of languages handled there), however our method as well as the threshold values are mainly inspired by their work. We first collected the hashes (Secure Hash Standard `sha` as computed and readily available by GitHub) for all the files (i.e. not only the metamodels) in the repositories, and computed the pairwise comparison of repositories using the Jaccard distance. For manual analysis of the individual cases where high similarity occurs, we collected samples (n=50 per range) from the subsets of similar repository pairs with similarities: 100%, 98-99.9%, 90-97.9%, 80-89.9% and 50-79.9%:

– 100% duplicated: mostly renamed repositories or ones redirecting to other repositories and ones cloned from Google Code and Eclipse.
– 98-99.9% duplicated: same projects with very few diffs (1-15 commits) cloned from Google Code, SourceForge or others from GitHub.
– 90-97.9% duplicated: same projects with some diffs (1-500 commits) cloned from Google Code, SourceForge, Eclipse; explicitly labelled as new versions, patches, or new features added.
– 80-89.9% duplicated: same projects but with significant change/customization.
– 50-79.9% duplicated: still same projects but with major changes.

5.2.3 Decision for Deduplication

Based on metamodel and repository duplication percentages, we follow the protocol below:

1. Given a repository pair, if metamodel duplication is 100% and repository duplication is >50%, we label the pair as duplicated. We cluster duplicated repositories using hierarchical clustering (see Babur et al. (2016) for an application of hierarchical clustering to metamodels) with a cutoff at 55% distance (threshold determined based on our preliminary exploration of 40 samples in the dataset). Per cluster, we keep a single representative repository and discard the rest. Based on our observations on the samples, we apply a simple technique to identify the *original* repository among the cluster (the earliest creation date of the repository) and include only that one. See Section 5.2.4 for a discussion on this strategy.
2. Given a repository pair, if metamodel duplication is <100% and repository duplication is > 50%, we consider the metamodels to be reused or repurposed in different contexts, which is interesting for our language usage study. While they may also be implicit versioned copies of the original repository (i.e. not explicit ones within the same repository or forks, both of which were excluded in the first place), we consider the delta large enough to include such repositories as additional data points. We include both repositories in our analysis.
3. If for any repository, metamodel duplication is > 90% and there is a large number of duplicated metamodels, this might signal a special case for metamodel collections (e.g. mined from other repositories on GitHub) and potentially not real development projects. We manually inspect the ones containing more than >50 metamodels (to keep the manual work feasible), and remove them from our dataset if they are indeed metamodel collections rather than engineered projects.

5.2.4 Results and Discussion

For item 1 in the protocol, we automatically detected 1,173 repositories involved in both significant metamodel and repository duplication. This is basically the union of the individual items in clone clusters found by hierarchical clustering with the 55%. Furthermore, by manually checking repositories with high number of metamodel duplication, yet *without* significant repository-level duplication (i.e. with respect to protocol item 3, signalling potential metamodel collections rather than engineered projects), we were able to detect 19 additional repositories to remove. This additional step we took proved to be useful, as there were significant cases: a single repository with a collection of 21k+ duplicate metamodels, other zoos and datasets with several hundreds of duplicate metamodels, unofficial forks and so on. As a result of removing those repositories and their metamodels, we eliminated a total of 38.351 metamodels from our dataset, 34,023 files remaining to be processed.

A note on deduplication we have performed is that it is a best-effort approach involving considerable amount of manual work. A more fine-grained approach would reveal cases, for instance, where our inclusion of representative repositories from duplicate clusters with respect to the earliest creation date is inaccurate. Manually inspecting random samples from our dataset, we could arrive at the following qualitative observations. Many cases in our dataset involved a major organization hosting a repository (e.g. Eclipse) and individuals unofficially forking and modifying the original repository (hence having a newer creation date). However, we have also observed duplicate repositories which contain *newer* versions, and therefore it makes more sense to include the most recent version, discarding the older ones.

We have adopted the general notion of repository-level duplication following the previous MSR literature (Lopes et al. 2017). We assume that when a metamodel is duplicated while the rest of the project is significantly different, the metamodel is reused *in a new context* (models, code and other artifacts). Therefore, we consider such reused metamodels as independent data points and did not deduplicate them. However, this distinction is a difficult one to be done precisely. For the repositories falling under category 1 (metamodel duplication 100% and repository duplication >50%) we performed an exploration of a sample of the repositories falling under category 1. We observed that those repositories were mostly *versions* or *slight variants* of each other rather than projects reusing the metamodel in a new context. A precise account of such cases is only possible after a separate and detailed cloning analysis, preferably not only covering the metamodels but also the models and other related artifacts. Note that when a metamodel is being used in other projects (even as-is), it is highly likely that the rest of the artifacts will vary, so the repository duplication will not be high. So such cases fall under category 2 in our protocol and we include them in our study.

Another point to make is that there is still significant file-level duplication, across the repositories as well as to some extent within the repositories. From our previous study, we also observed higher types of clones (e.g. near-miss clones, where a few modifications exist between clone pairs while they remain to be highly similar) among the metamodels (Babur et al. 2019). Given the scope of this study, we do not investigate the cloning phenomenon further.

Finally, we report that after the deduplication step, our dataset contains 19.6k unique metamodel files in total (out of 34k metamodels over different repositories; see Section 5.2 on why we keep duplication in our dataset). Given the 72.3k metamodels that remained after filtering out the non-engineered repositories, this means we have a 73% duplication overall 19.6k unique models correspond to 27% of the the whole dataset of 72.3k. This percentage of duplicate EMF metamodels on GitHub seems to be much higher than what is reported by Hebig et al. for UML models as 12% (Hebig et al. 2016). Some part of this can be attributed to the existence of large collections of metamodels mined from GitHub, yet stored again on GitHub projects (see discussion above), as well as the fact that Ecore and UML reside in different meta-levels (cf. MOF hierarchy in Section 2.2). We leave it as future work to investigate this phenomenon in detail.

5.3 Filtering Out Unprocessable Files

As the final step of filtering, we eliminated the metamodels which we could not process due to one of the following reasons: (1) unparseable by the EMF parser (e.g. not well-formed, or dependent on and using metaclasses from other metamodels), (2) unprocessable by the SAMOS extractor. We believe the part of (1) on dependence on other metamodels needs further clarification. Metamodels can import other metamodels beyond basic Ecore, in the sense that they can reuse the metaclasses of the imported metamodels as building blocks in the language specification of the importing one. Given the large GitHub ecosystems, we performed a manual investigation on a sample set of metamodels. In that sample, we could not track all the imported metamodels in on GitHub and successfully resolve them. And even when we could find them, doing this for all our dataset would complicate the processing and analysis. Therefore, we opted to include only the metamodels built only based on Ecore as opposed to importing other metamodels. We however included cases where there are proxy references to other metamodels (see Steinberg et al. (2008) for the details on proxy references), which can remain unresolved without affecting our studies.

Performing this step, we identified 1,191 metamodels as unprocessable and excluded them from our final dataset: 1,142 metamodels involving parser error and 49 unprocessable by SAMOS. A brief inspection of a sample of those unprocessable files reveals serialization issues and possibly cyclic dependencies causing the parser to time out. This concludes the filtering phase, and we move on to extraction with 32,832 metamodels left.

6 Extracting and Querying Information

We use and extend SAMOS to extract the necessary information for analysis, in a format to be imported into Neo4j. As outlined in Section 2.3, SAMOS is already capable of traversing the underlying graph of EMF metamodels and generating chunks of node and edge information in the form of bigrams (i.e. n-grams with two nodes and a connecting edge) (Babur and Cleophas 2017). To acquire the metadata about the metamodels (e.g. creation date) and the repositories (e.g. total # commits) we added to SAMOS scripts making use of the GitHub REST API.

In order to serialize the bigrams and other information in csv format, we implemented an export functionality in SAMOS. The resulting series of csv files are then imported into Neo4j using the Cypher shell[18], a command-line tool for data import/export and querying for Neo4j using the Cypher query language. Eventually, all the information relevant for our analysis is stored in Neo4j as graph data, as demonstrated in Fig. 7.

6.1 Using Cypher Queries and R Scripts in Combination for Analysis and Visualization

We formulated our research questions as Cypher queries where possible for a clean and reproducible analysis process, while utilizing R scripts for implementing additional logic. We executed the queries via R using the Neo4j bridge, and eventually obtained (and post-processed where needed) the results as well as the visualizations in R. We performed the analyses on a MacBook Pro with 2,4 GHz Intel Core i9 CPU and 32 GB 2400 MHz DDR4 RAM. The importing took ∼6 hours of time, while the analyses were completed in a total of 3.5 hours.

6.2 Data Availability

We provide a list of metamodel files included in the analyses, as well as the R notebook documents (containing the Cypher scripts) as supplementary material[19]. The Neo4j database dump is available upon request to enable further research on our dataset. It is not possible to share the metamodel files and content publicly due to licensing issues and GitHub terms of service[20].

[18] https://neo4j.com/docs/operations-manual/current/tools/cypher-shell/

[19] https://www.win.tue.nl/~obabur/publications/EMSE22

[20] https://docs.github.com/en/github/site-policy/github-terms-of-service

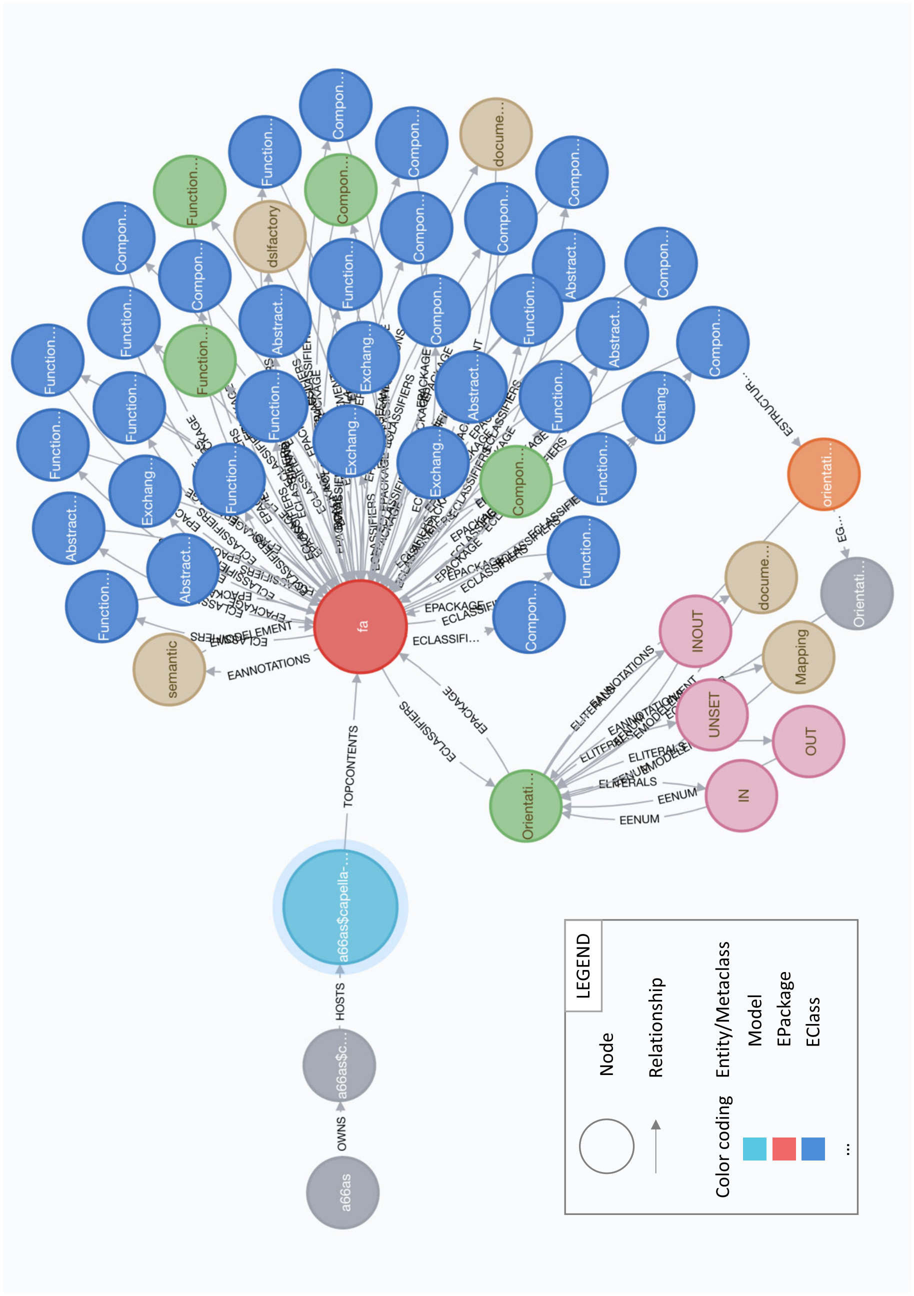


Fig. 7 An example chunk of metamodel content and metadata visualized as a graph in Neo4j

7 Data Description / Raw Statistics

In this section, we provide descriptive statistics of the final dataset used in the subsequent analyses in Section 8.

Table 4 Statistics on relations among the number of owners, repositories and metamodels

	min	median	mean	max
repo per owner	1	1	1.48	75
metamodel per repo	1	2	8.02	7,483
metamodel per owner	1	2	11.87	7,483

7.1 Repository Statistics

Our dataset consists of 4,091 repositories belonging to 2,767 owners, and containing a total of 32,832 metamodels. In an overwhelming majority of the cases, there are very few (1-2) repositories per owner as well as very few metamodels per owner and repository, with further extreme outliers. Table 4 summarizes the key statistics. Owners hosting the largest number of repositories include `eclipse` (the Eclipse Foundation[21]) and `atlanmod` (NaoMod Research Group from IMT Atlantique, France[22]). The extreme outlier in terms of the number of metamodels per repo as well as metamodels per owner (both 7,483) is `damenac/puzzle` hosting the puzzle toolset.

Subscriber, watcher and fork counts are among widely used for assessing the popularity of a GitHub repository (Cosentino et al. 2017). We present these in Table 5. It seems most repositories have very low popularity, with a few standing out: e.g. `puppetlabs/puppet-specifications` in terms of subscribers, `geotools/geotools` of subscribers and watchers, `opensourceBIM/BIMserver` for all three measures. The distribution of these measures roughly follow the power law, as reported in the literature, to hold for GitHub projects in general (Cosentino et al. 2017).

While the programming language reported by GitHub can be inaccurate at times (attributed to the underlying software *Linguist*[23]), we find it noteworthy to report that a large majority of the repositories (2,790 out of 4,091) have Java as the main language. This is more than twenty times more than the second most popular programming language, HTML (106); and the third one, Xtend, being a model transformation language (96). The dataset includes a total of 58 distinct languages reported as the main language.

7.2 Repository and Metamodel Trends

We report the trends with respect to the number of created repositories per year and the repository age (i.e. the number of years since the last update) as well as the same metrics for the individual metamodels. Besides merely reporting raw statistics, we compare the trends observed with our earlier findings, where we observed a clear trend in increasing number of metamodels on GitHub (Babur et al. 2017; Babur 2019). The number of created repositories and metamodels per year are outlined in Fig. 8. Thanks to the extensive filtering and deduplication we have done in this paper, we cannot confirm a clear upwards trend for neither of these numbers. The trend reported previously can be partly explained by either cloning of

[21] https://www.eclipse.org/

[22] https://www.atlanmod.org/

[23] https://github.com/github/linguist

Table 5 Statistics on popularity measures for the repositories

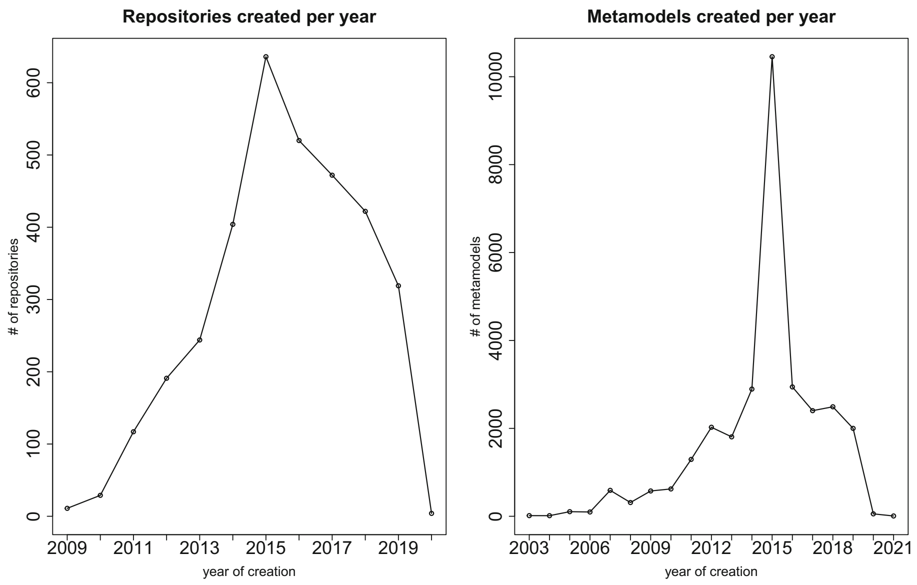
	min	median	mean	max
subscribers	0	1	3.83	186
watchers	0	0	4.03	959
forks	0	0	2.48	868

repositories/metamodels or the increasing population of non-engineered metamodel repositories on GitHub. However, there can be other factors affecting this phenomenon, the study of which is out of scope for this paper. There is an exceptional spike in the number of created metamodels in 2015, which can mostly be attributed to the repository `damenac/puzzle`, which has 7,483 metamodels as discussed in Section 7.

Finally, we report the age of the repositories and metamodels in Fig. 9. The numbers indicate a varied distribution of ages, and not too many *old* entities, meaning they have been inactive for a long time. The spike in the number of metamodels with age 5 was previously explained, i.e. for the spike of the metamodel count in 2015 mostly owing to the repository `damenac/puzzle`.

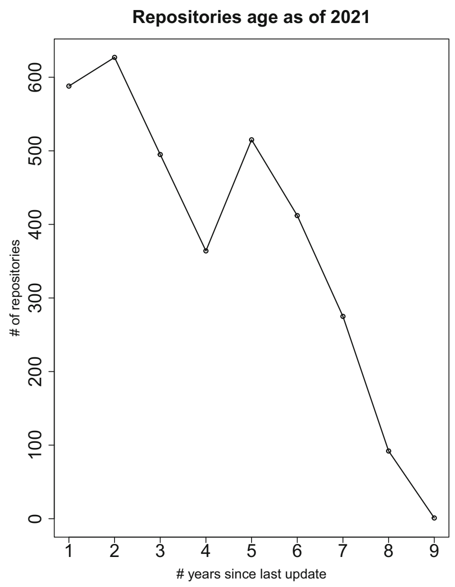
7.3 Metamodel Statistics

Next, we discuss the general statistics in terms of the metamodels, as depicted (with log transformation) in Fig. 10. The 32,832 metamodels in our dataset have their corresponding file serialization with a wide range of size: from 103 bytes to ∼10 megabytes. This is interesting not only for purely descriptive purposes, but also exposes limitations in using various search

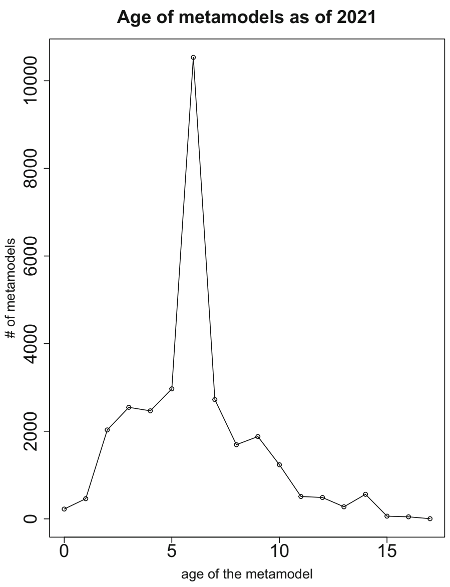


(a) repositories created per year (b) metamodels created per year

Fig. 8 Statistics on repository and metamodel creation over the years



(a) repository age



(b) metamodels age

Fig. 9 Statistics on repository and metamodel age as of 2021

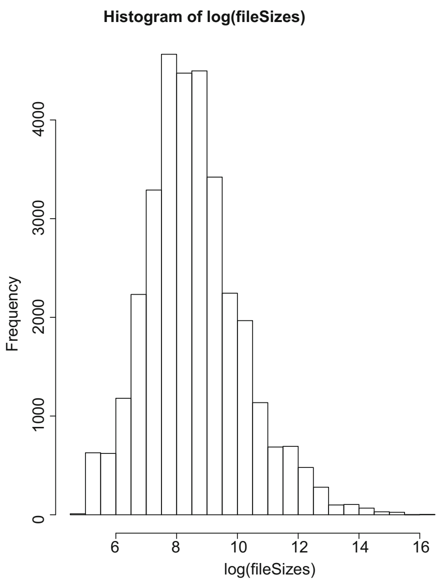
methods. The distribution is roughly log-normal, and the large numbers in the upper part of the distribution have implications for adopting a mining method: there are indeed large files which would be impossible to retrieve by the GitHub code search API (see Section 4.2). The dataset in total contains more than 12 million model elements and 9 million relationships. We also checked the distributions for the file sizes, as well as metaclasses and relationships per metamodel. Our hypothesis was that they might follow a log-normal distribution, as similar results have been reported for other software engineering artifacts, e.g. source code file sizes (Concas et al. 2007). We executed an Anderson-Darling test for normality on the log-transformation of these three metrics, and observed a very low p-value ($< 2.2 \times 10^{-16}$). We therefore conclude they are not log-normally distributed. Manually inspecting the Cullen and Frey graph revealed gamma or beta distributions as most likely estimations. We leave a more rigorous estimation of the distributions as future work. The largest metamodel for Model-Driven Health Tools[24] contains 50k+ metaclasses (i.e. counting all types of metaclasses —see Section 2.2 — ranging from EClasses and EPackages to EOperations and EParameters) and 124k+ relationships, which is considerably large but not to the extent as reported in the literature for very large models with millions of model elements (Pagán et al. 2011). Among the largest models are the Fast Healthcare Interoperability Resources[25] metamodel, test and benchmark metamodels for the model transformation language Acceleo[26], and the EastADL[27] architectural description language.

[24] https://github.com/mdht/mdht-models

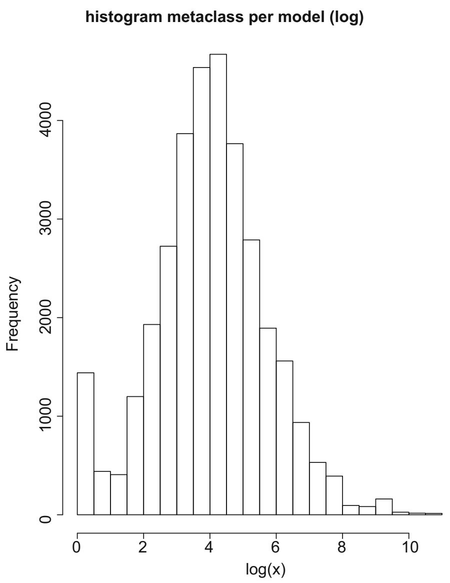
[25] https://www.hl7.org/fhir/overview.html

[26] https://www.eclipse.org/acceleo/

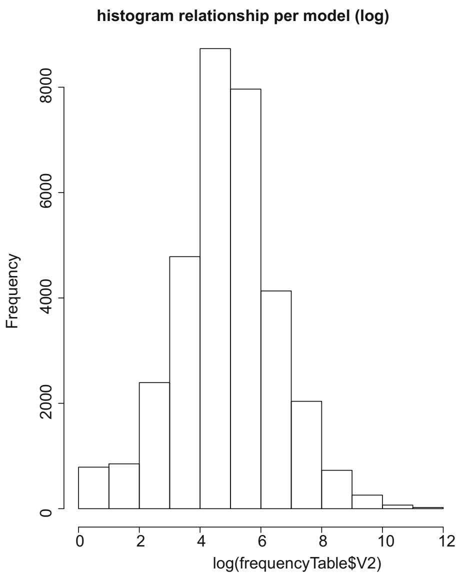
[27] https://www.east-adl.info/



(a) file sizes



(b) metaclass per mm



(c) relationn per mm

Fig. 10 Statistics on metamodel sizes, metaclasses and relations

7.4 Metamodel Domains

We were not able to perform a repository-level domain analysis as suggested in the literature (Ray et al. 2014). A large portion of the repositories we scanned had either no readme files, or very limited and non-descriptive content in them. Instead, we performed a metamodel-level domain analysis using SAMOS (Babur et al. 2016). We extracted the identifier names from the metamodels and preprocessed the names using natural language processing (including tokenization, lemmatization). We then built a document term matrix (term frequency, inverse document frequency approach where we consider a metamodel as a *document*, see Babur et al. (2016)). Note that we discarded common English language stop words, as well as very short or meaningless tokens (such as *A* or *n1*). On top of the matrix, we obtained pairwise cosine distances and computed the clusters via hierarchical clustering with cut-off distance of 20%. We manually checked the large clusters (size>50) to see whether we could identify major domains or patterns. We have not performed an additional step of validating the results in this study due to the fact that SAMOS with its clustering functionality has already been validated on previous publications (Babur et al. 2016, 2019, 2022).

Table 6 depicts the results. On the one hand, we can see clusters of major modeling languages such as Ecore(#139), UML(#157), BPMN(#150); and similar languages such as library management DSLs(#250), the manual inspection of which reveals that they are used mostly for experimentation/demonstration/education purposes. Note that the latter cluster implies these might be non-engineered metamodels in engineered repositories, as well as non-engineered repositories which were not correctly classified by PHANTOM. We also see Java metamodels(#162), possibly to be used for forward engineering as well as model-driven reverse engineering and analysis. On the other hand, we see groups of small models with dummy or test content(e.g. #143, #146 and #182), the clustering of which is less meaningful due to their limited content. Interestingly, there are further metamodels from a single reposi-

Table 6 Large metamodel clusters in terms of domains, the total number of metamodels in each cluster and the manual description of the cluster

cluster ID	# metamodels	description
135	73	state machine DSLs
139	210	Ecore meta-metamodels
143	89	small ’test’ metamodels
146	76	small ’class’/abstract syntax tree metamodels
150	92	Business Process Modeling & Notation (BPMN)
157	115	UML metamodels
162	127	Java metamodels
182	96	dummy DSLs e.g. with simple classes ’Hello’, ’Greeting’
250	74	library management DSLs
401	237	small metamodels for testing transformations, keyword ’root’
757	104	Atlas Transformation Language metamodels
765	66	XML metamodels
1,009	55	type language DSLs
5,441	168	test DSLs from a single repo, model2doc transformation
5,486	110	dummy metamodels, common keyword ’package’
6,045	139	topology control analysis metamodels group 1
6,046	111	topology control analysis metamodels group 2
7,482	69	Yakindu statechart language variants

tory, where analysis cases are specified as input metamodels for model transformation in the case of evaluating topology control algorithms[28] (#6,045 and #6,046). Overall we observe a heterogeneous dataset with metamodels being used for a variety of purposes. A more rigorous characterization of the metamodel domains is left as future work.

8 Empirical Analyses on Language Usage

In this section, we report the various —quantitative as well as qualitative— empirical analyses we have performed on the language usage. The overarching research question, following our goal for doing this study, is as follows: how is the EMF metamodeling language used by public engineered projects on GitHub? We adopted the research questions posed by Herrmannsdoerfer et al. (2010), while tackling additional relevant in the scope of our study. Each subsection tackles a different aspect of language usage, therefore is formulated with its own motivation, (sub-)research questions and discussion.

8.1 Metaclass Usage

The metaclasses represent major building blocks of the language represented by the metamodel. It is a logical first step in our language usage analysis to study the number of metaclass instances and their distributions over the metamodels. Such a study might reveal over- and underuse (with respect to a baseline, to be discussed in each subsection) of certain metaclasses, and lead to suggestions for the language design —e.g. removing a highly underused metaclass, or extending an overused one—. Such suggestions contribute to a simple, refined and focused language design as reported in the literature among good practices (Paige et al. 2000). We extensively compare our results with Herrmannsdoerfer et al. (2010), who performed a similar analysis on a different dataset. A major point when comparing the two datasets is the magnitude: our dataset contains 32k metamodels with 12 million model elements, while Herrmansdoerfer et al.'s contains 2k metamodels with less than a million model elements.

We identify the following research questions to be investigated in this subsection. Our motivation for these two research questions is to detect the over- and underuse of metaclasses.

- **RQ1.1** What metaclasses are the most used (in total as well as distribution over the metamodels)?
- **RQ1.2** What metaclasses are rarely used or not used at all (in total as well as distribution over the metamodels)?

We present our quantitative results distributed over separate subsections and tables. In Table 7, we show the total number of metaclasses and their percentages cumulatively in the whole dataset, with our results on the left-hand side, and Herrmannsdoerfer et al.'s on the right-hand side. We depict a binary categorization of presence or absence of a metaclass in metamodels in Table 8 as it might paint a different picture than total count percentages (which it does, as we will see as follows). Finally we discuss a subset of distributions for some metaclasses.

[28] These seem to be related to Cobolt - A model-based tool for evaluating topology control algorithms by eMoflon: https://github.com/eMoflon/cobolt.

Table 7 Metaclass usage counts and overall shares

Our dataset				Dataset of Herrmannsdoerfer et al. (2010)			
#	metaclass	count	share	#	metaclass	count	share
1	EGenericType	3,388,307	27.69%	1	EString..Entry	328,920	44.51%
2	EString..Entry	2,995,124	24.48%	2	EGenericType	141,043	19.09%
3	EAnnotation	1,840,180	15.04%	3	EAnnotation	102,863	13.92%
4	EReference	1,078,387	8.81%	4	EReference	53,177	7.20%
5	EAttribute	1,007,981	8.24%	5	EClass	41,506	5.62%
6	EClass	850,237	6.95%	6	EAttribute	36,357	4.92%
7	EEnumLiteral	389,374	3.18%	7	EEnumLiteral	10,643	1.44%
8	EParameter	262,178	2.14%	8	EOperation	7,158	0.97%
9	EOperation	236,957	1.94%	9	EParameter	7,060	0.96%
10	EPackage	64,282	0.53%	10	EPackage	4,530	0.61%
11	EEnum	61,867	0.51%	11	EDataType	2,747	0.37%
12	EDataType	54,408	0.44%	12	EEnum	2,513	0.34%
13	ETypeParam.	5,665	0.05%	13	ETypeParam.	226	0.03%
14	EObject	52	∼0%	14	EObject	0	0%
15	EFactory	0	0%	15	EFactory	0	0%

Note that *EStringToStringMapEntry* is shortened as EString..Entry, *ETypeParameter* as ETypeParam

8.1.1 Most Used Metaclasses Overall

Results The most used metaclasses overall can be found in the top rows in the Table 7, which orders the metaclasses with respect to their total counts and shares. Our results are shown on the left, while Herrmannsdoerfer et al. 's are on the right side of the table. *EGenericType*,

Table 8 Metaclass presence, in terms of total number and shares over the metamodels

#	metaclass	present	absent	share
1	EPackage	32,765	67	∼1.00
2	EClass	31,697	1,135	0.97
3	EGenericType	30,894	1,938	0.94
4	EAttribute	28,369	4,463	0.86
5	EReference	28,098	4,734	0.86
6	EDataType	27,509	5,323	0.84
7	EAnnotation	11,492	21,340	0.35
8	EStringToStringEntryMap	10,875	21,957	0.33
9	EEnum	10665	22,167	0.32
10	EEnumLiteral	10,570	22,262	0.32
11	EOperation	5,759	27,073	0.18
12	EParameter	4,326	28,506	0.13
13	ETypeParameter	1,010	31,822	0.03
14	EObject	30	32,802	∼0
15	EFactory	0	33,471	0

EStringToStringMapEntry and EAnnotation are the three most used metaclasses overall, also confirming the findings of Herrmannsdoerfer et al. even if with different shares and ranking. In our results, *EStringToStringMapEntry* has a lower rank (i.e. second place rather than first) and a much lower share (27,69% rather than 44,51%). For *EGenericType* and *EAnnotation*, the differences in the shares is less significant: correspondingly 27,69% vs 19,09% and 15,04% vs 13,92%.

Discussion The results might seem unintuitive in the beginning, as one would expect conceptually more central building blocks of the language (e.g. *EClasses* and *EAttributes*, mapping to the main domain concepts and their properties) to be the most prominent. However, we can explain this situation considering the fundamental language design. *EGenericTypes* form the basis of all type definitions (generic *AND* non-generic), therefore are associated to all the typed metaclasses in the language. At least one *EGenericType* instance is associated to every single typed element instance. For example, for each of the two *EAttributes* having an *EString* type (i.e. a standard built-in type in Ecore), a separate *EGenericType* (referring to the actual *EString* type) is created and associated to each *EAttribute* instance. For particular (e.g. parametrized) generic type definitions, on the other hand, the number of *EGenericType* instances are further inflated due to their nested representation containing multiple *EGenericTypes*. It is worthwhile to note that EMF employs two techniques to hide the underlying complexity of the *EGenericType* hierarchy. For the non-generic types (and the sake of backwards compatibility pre-EMF 2.2), users are allowed to specify regular types without using *EGenericTypes* in the concrete XMI representation and graphical editor. Those are nevertheless internally transformed into *EGenericType* instances in the abstract syntax. Additionally, the complex generic type hierarchy is hidden in the Ecore Editor user interface. The type, instead, is simply displayed with a shorthand text notation. While this might involve the internal representation of the language and look superficial, we believe it is an important note for our study which focuses on the language syntax.

EAnnotations can be attached to any model element and used for a variety of purposes: documentation, language extension constraints, mappings, code generation directives and so on. *EStringToStringMapEntry* metaclasses, in turn, represent key-value pairs within *EAnnotations*, which are modelled as first-class entities. Thus a simple map-type with several key-value pairs is syntactically represented as a tree with a separate instance of EStringToStringMapEntry for every single key-value pair. This explains the overrepresentation of *EStringToStringMapEntry* in the dataset overall.

As a final note, the different order in the top three most used metaclasses compared to the results by Hermannsdoerfer et al. can possibly be attributed to our dataset having more of a 'out-of-the-wild' nature, containing more ad hoc metamodels with no proper documentation or code generation attached. Nevertheless, we believe it is interesting to see our results being very similar to Hermannsdoerfer et al., while we followed a state-of-the-art MSR study with a rigorous protocol (notably in data collection and filtering) and covered a much bigger population.

Suggestions Given the underlying language design in this current state, *EGenericTypes* naturally exist as a highly frequent metaclass. The generics mechanism in Ecore is already designed with the goal of simplicity (cf. one of the goals in designing modelling languages (Paige et al. 2000)), e.g. compared to that of UML, which is *more verbose*[29]. Another

[29] https://www.eclipse.org/modeling/mdt/uml2/docs/articles/Defining_Generics_with_UML_Templates/article.html

hypothesis we make is that this bloated structure of nested EGenericType instances might cause performance issues when dealing with very large models, e.g. consisting of millions of model elements (Kolovos et al. 2013). A possible modification to the language would be having two separate and explicit type metaclasses, one for regular and one for generic types. This would separate the two types more clearly for the sake of understandability and maintainability of the language. This way the metamodel is also better matched to how the language is presented to the user via the editors, which hide the generics features by default for non-generic types.

As for *EStringToStringMapEntry*, rather than being first-class citizens as metaclasses, they can be modeled as *Dictionary* data types (as e.g. commonly used in some programming languages including Python). Though, this means that this new data type should be added to the Ecore meta-metamodel. The creators explicitly state in the EMF reference book: “the use of map-typed features. Although there is no map type explicitly modeled in Ecore, support for such features is enabled through a special behavior of the code generator”(Steinberg et al. 2008). We believe having a map type already in the language level conceptually suits better to the represented data and avoids workarounds in the tooling or code generators, while keeping the same level of expressivity in the language.

EAnnotations are used both in free format (for arbitrary form of documentation), and in very specific patterns (for code generation). The EMF editor recognizes some of the patterns and provides some level of guidance for the user, e.g. in the form of a code generation icon. A possible language extension would be specializing *EAnnotations* to capture the most frequent patterns, therefore providing language-level structure and validation to the use of those patterns. We discuss this in more detail in Section 8.4.1.

8.1.2 Most Present Metaclasses per Metamodel

Results The metaclass presence statistics are given in Table 8. The most present metaclasses can be found in the top rows. Note that this table gives a different picture than the metaclass share overall in Table 7. *EPackage* and *EClass* are present in an overwhelming majority ($> 97\%$) of the metamodels. *EGenericType* is also highly frequent (94%), followed by *EAttribute*, *EReference* and *EDataType* with $> 84\%$ frequency.

Discussion The conceptually more central building blocks for a metamodel are the most frequent metaclasses in terms of presence: *EPackage* for structuring, *EClass* for domain concepts, *EAttribute* and *EReference* for parametrizing those concepts, *EDataType* for domain data types, and finally *EGenericType* representing type information.

Suggestions We revisit our suggestions above on EAnnotation and EStringToStringEntryMap. While we confirm the same suggestions, it should be underlined that those metaclasses are mostly gathered within a certain group of metamodels (i.e. only some — $< 35\%$— metamodels have them, but they have a lot of them), that are e.g. well-documented, or targeted for code generation.

8.1.3 Not or Least Used Metaclasses Overall

Results The not or least used metaclasses overall can be found in the bottom rows in the Table 7. As the table depicts, we have not found any *EFactory* metaclass in our final dataset, except a few which we manually identified in the filtered out metamodels (due to being unparseable or having external dependencies). *EObjects*, while being non-zero in contrast to Hermannsdoerfer et al., are also virtually absent in our results. The next set of the least used metaclasses are *ETypeParameter*, *EDataType*, *EEnum* and *EPackage* with $< 1\%$; and *EOperation*, *EParameter* and *EEnumLiteral* with $< 4\%$.

Discussion *EFactory* is an auxiliary metaclass for creating instances of metamodel elements, therefore it typically does not appear in metamodels as part of the language abstract syntax; it is rather created and used in run time (e.g. programmatically creating models or generating code). EFactory is not even exposed to the user through the Ecore graphical editor. The few *EObject* instances, which we manually inspected, turned out to be potentially erroneously created ones within *EAnnotation* objects (completely absent in the serialized metamodels, however they appear in run time when EMF loads the metamodel). The low share of *ETypeParameter*, on the other hand, indicates the lack of use of the generics feature of Ecore. For *EDataType*, *EEnum* and *EEnumLiteral*, the few instances overall are expected to a great extent. While these metaclasses are essential parts of the language design, there are few of them; being specified once and used in many places in the language. A similar argument holds for *EPackage*: following the design of the underlying domain concept for *EPackage* (i.e. merely a container for collecting domain concepts, see Section 2), only a few instances per metamodel likely suffice. Finally, the relative lack of *EOperation* and *EParameter* in the metamodels in our dataset indicate a more structural or ontological nature of the metamodels (representing domain concepts and their relations), rather than operational details.

Suggestions Herrmanssdoerfer et al. suggests changes on two of the items: "*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, disallowing its instantiation. A class should be transient if its instances are not expected to be made persistent—such a class does not represent a language construct. For instance, in Ecore, EFactory—a helper class to create instances of the specified metamodel-should be transient.*" In the Ecore meta-metamodel, EFactory and EObject are both defined as concrete classes, so the suggestions on the metamodel-level might hold at first glance. However, EObject might not be made abstract in practice due to its important role for instantiation in dynamic EMF, and EFactory cannot be made transient as it is allowed for attributes rather than classes in EMF. Note that, in addition, in the Java implementation of EMF, EFactory and EObject are both designed as interfaces. Therefore, in the Java implementation the situation can be different, and can have implications affecting, e.g. , the further operation of dynamic EMF. We leave it as future work to investigate this part in tandem with language architects. As another rarely used metaclass, we see in our results *ETypeParameter* has a very low share (0.05%). Nevertheless, *ETypeParameter* as a part of the generics package, represent a major feature of the language, and we cannot suggest their removal altogether. The rest of the rare features, as will be elaborated in the next section, are present in a significant share of metamodels ($> 10\%$), therefore should not be removed or made transient.

8.1.4 Not or Least Present Metaclasses per Metamodel

Results Similar to the overall shares reported in Table 7, *EFactory* and *EObject* metaclasses have zero/near-zero shares, while *ETypeParameter* is also quite rare (0.03%). The rest of the overall rare features (*EDataType*, etc.) are present in a significant share of the metamodels ($13 - 32\%$).

Discussion These results give a complementary picture with the overall share. The least used features are also the least present per metamodel (*EFactory*, *EObject*, *ETypeParameter*). The metaclasses representing the core building blocks, however, are as expected present in a significant share of metamodels.

Suggestions We confirm the suggestions proposed above when discussing the overall share of the metaclasses.

8.2 Metaclass Feature and Relationship Usage

Features are the core elements in Ecore, along with *EClass*, for defining the concepts and their relation in the language. Structural features allow parametrizing the concepts (*EAttribute*) and relating them to each other (*EReference*), while behavioral features allow specifying the operational aspects and their parametrization (*EOperation* with *EParameter*). Features, however, are in the end just a particular type of relationship. The Ecore language consists of a wide range of other relationships to shape up the language abstract syntax: containment, inheritance and various association relationships. It is worthwhile to investigate the usage of features and relationships, as their lack of use or underuse (in terms of multiplicity) could indicate room for possible language improvements for Ecore, e.g. in the form of removal or changing the lower/upper bounds (Herrmannsdoerfer et al. 2010) (e.g. changing a one-to-many relationship into one-to-n or even one-to-one). We identify the following research questions to be investigated in this subsection. Our motivation for these research questions is to detect the lack of use or underuse of features and relationships.

– **RQ2.1** What features/relationships are not used, or rarely used?
– **RQ2.2** What features/relationships are not used to their full multiplicity?

We present our quantitative results in Tables 9 and 10, as feature/relationship usage per metaclass.

8.2.1 Least Used Features and Relationships

Results Considering the statistics in Table 9, we see most of the features/ relationships have low medians (ranging between 0 and 5). The presence table, however, gives a better picture on the usage. We can see from the Table 10 that the following relationships are rarely ($< 5\%$) used: *eTypeArguments*, *eLowerBound* and *eUpperBound* for *EGenericType*, *contents* for *EAnnotation*, *eKeys* for *EReference*, *eGenericExceptions* for *EOperation*, and *eTypeParameter* for *EClassifier*. The subpackaging relationship (*eSubPackages*) as well as the behavioral structural type (*eOperations*) and references for *EAnnotations* are also only occasionally ($< 10\%$) used.

Discussion As previously observed, generics are rarely used in our dataset, hence explaining most of the corresponding relationships in this section being rare. As an example, *eTypeArguments* indicates a complex (parametrized) generic type, and may not be generally preferred by an average modeler.

Secondly, the relationships *eKeys* and *contents* represent very specific aspects of the language: explicitly defining the key elements to be used in the serialization of an *EClass*, and

Table 9 Statistics for features and relationships in the metamodels

	feature/relationship	min	median	mean	max
1	EClass eStructuralFeatures *	0	1	2.45	581
2	EClass eOperations EOperation	0	0	0.27	496
3	EOperation eParameters EParameter	0	1	1.11	13
4	Model topContents *	0	1	1.09	95
5	EPackage eClassifiers *	0	5	15.02	2,015
6	EPackage eSubpackages EPackage	0	0	0.45	1,624
7	EOperation eGenericExceptions *	0	0	0.02	6
8	EClass eGenericSuperTypes *	0	1	0.82	33
9	EEnum eLiterals EEnumLiteral	0	4	6.29	8,393
10	EReference eOpposite *	0	0	0.20	1
11	EReference eKeys *	0	0	∼0.00	4
12	EGenericType eClassifier *	0	1	0.96	1
13	EGenericType eTypeArguments EGenericType	0	0	0.06	10
14	EGenericType eLowerBound EGenericType	0	0	0.00	1
15	EGenericType eUpperBound EGenericType	0	0	∼0.00	1
16	ETypeParameter eBounds EGenericType	0	0	0.27	3
17	EAnnotation contents *	0	0	0.01	74
18	EAnnotation references *	0	0	0.05	9
19	* eAnnotations eAnnotation	0	0	0.24	92
20	EClassifier eTypeParameters ETypeParameter	0	0	0.01	10
21	ETypedElement eGenericType *	0	1	0.97	1
22	EAnnotation details EStringToStringMapEntry	0	1	1.63	142

The asterisk in some of the rows means there are multiple metaclasses which could be in the relationship. For instance, in row 6, *EPackage* can have a set of *EClass*, *EDataType* and *EEnum* instances as its *eClassifiers*

embedding model elements in an *EAnnotation*. A conjecture about the underuse of the former is that modeling exceptions explicitly might be a too low-level activity for the users, who are not so often modeling the behavioral features in the form of *EOperations* in the first place. As for the latter, it seems to be the case that *EAnnotations* are occasionally used (19%) but presumably for simple purposes such as documentation, without embedding model elements. Note that in slightly more cases (5%), users seem to be using references to model elements (via *references*) rather than containing the elements directly.

Suggestions We suggest the deprecation and eventually removal of the *eKeys* relationship, as well as the restriction of using contents for *EAnnotation*. Removing the generics package completely would limit the language to the few cases where they are needed and used. So even if it would lead to a significant simplification in the language, we do not suggest its removal.

8.2.2 Multiplicity Usage for Features and Relationships

Results From Table 9, we see that for a few of the features, which have [0..1] multiplicity (see the metamodels in Figs. 1 and 2), we have the proper minimum usage as 0 and maximum as 1;

Table 10 Presence for features and relationships

	relationship	present	absent	percentage
1	EClass eStructuralFeatures *	606,607	243,630	0.71
2	EClass eOperations EOperation	57,128	793,109	0.07
3	EOperation eParameters EParameter	148,305	88,652	0.63
4	Model topContents *	32,809	23	∼1.00
5	EPackage eClassifiers *	59,976	4,306	0.93
6	EPackage eSubpackages EPackage	4,888	59,394	0.08
7	EOperation eGenericExceptions *	3,749	23,3208	0.02
8	EClass eGenericSupertypes *	608,099	242,138	0.72
9	EEnum eLiterals EEnumLiteral	61,248	619	0.99
10	EReference eOpposite *	21,8805	859,582	0.20
11	EReference eKeys *	1,815	1,076,572	∼0.00
12	EGenericType eClassifier *	3,258,615	129,692	0.96
13	EGenericType eTypeArguments EGenericType	97,245	3,291,062	0.03
14	EGenericType eLowerBound EGenericType	34	3,388,273	∼0.00
15	EGenericType eUpperBound EGenericType	283	3,388,024	∼0.00
16	ETypeParameter eBounds EGenericType	1,479	4,186	0.26
17	EAnnotation contents *	12,156	1,828,024	0.01
18	EAnnotation references *	89,820	1,750,360	0.05
19	* eAnnotations EAnnotation	1,840,023	7,836,861	0.19
20	EClassifier eTypeParameters ETypeParameter	5,665	1,199,090	0.01
21	ETypedElement eGenericType *	2,500,130	85,373	0.97
22	EAnnotation details EStringToStringMapEntry	1,680,525	159,655	0.91

see for instance *eOpposite*, *eClassifier* and *eLowerBound*. For the majority of the remaining features, which have [0..$*$] multiplicity, we have a minimum usage of 0 and a maximum usage of n$>$ 1 and going to the order of a few thousands.

Discussion The feature usage is consistent with the specified multiplicities in the metamodel, pointing to an accurate design from the language end in line with its usage and correct implementation in the tooling. Therefore, we do not suggest any changes in the language based on our study.

8.3 Metaclass Attribute Value Usage

The possible values of an attribute instance are dictated by the type associated with that attribute. It is interesting to compare the theoretical/mathematical value range with the actual range and distribution of values used in metamodels. We aim to investigate the non-derived (see description in Section 2.2) attributes only, as derived attributes are secondary sources of information, as the name implies. First, we expect to see valid values in the allowed theoretical range. Secondly, we expect a balanced distribution of values in usage fitting in and spanning across the theoretical range. An unbalanced distribution would signal (a) a potentially wrong choice of the attribute type or (b) an unnecessary or unused attribute in the first place. An example for (a) would involve using a handful of distinct values for the virtually infinite range of the Integer type, which could signal e.g. specializing the metaclass or using an enumeration type instead. As for case (b), using one of the boolean values extremely frequently (i.e. close to 100%) might signal that the attribute is not used in practice and might be removed. For the remaining cases, we are interested in the most frequently used values, as they can have implications for the default values built into the language and API/tool support. We underline the distinction between having default values on the language level versus through API or tool support. An attribute can have a default value directly on the language level (e.g. see *ETypedElement.ordered* set to true in Fig. 1). Attributes can also have default values in the EMF editor, e.g. when creating a new *EClass* its *abstract* attribute is set to false by the tool, or even in the Java API level, e.g. when loading a serialized Ecore metamodel into memory, a non-existing *abstract* attribute for an EClass instance is set to false. In the end, having attributes with appropriate types and default values might potentially help modelers to correctly and efficiently build their metamodels. Using more restricted types and avoiding redundant attributes further benefit the language designer, e.g. in terms of reduced testing and validation effort.

We identify the following research questions to be investigated in this subsection.

- **RQ3.1** What attribute values are not used, rarely used or used outside their theoretical range?
- **RQ3.2** Which attributes (already having default values) do not have the most used value as default value?
- **RQ3.3** Which attributes have most often used values?

We decompose our results with respect to the attribute domain: boolean, integer and string, depicted correspondingly in Tables 11, 12 and 13.

Table 11 Statistics and information on an illustrative set of boolean attribute values

	attribute	true	false	mode	default (metamodel)	default (API/tool)
1	EReference.containment	54%	46%	true	NA	false
2	EAttribute.required	36%	64%	false	NA	NA
3	EReference.required	26%	74%	false	NA	NA
4	EAttribute.transient	10%	90%	false	NA	false
5	EAttribute.unique	90%	10%	true	true	true
6	EDataType.serializable	93%	7%	true	true	true
7	EAttribute.iD	1%	99%	false	NA	false
8	EReference.unique	99%	1%	true	true	true
9	EEnum.serializable	99.92%	0.08%	true	true	true
10	EOperation.unique	99.42%	0.58%	true	true	true
11	EParameter.unique	99.73%	0.27%	true	true	true

The last two columns indicate the default values (if any) expressed directly in the metamodel, and applied in practice by the Java API or tool support

Table 12 Statistics and information on an illustrative set of integer attribute values

	attribute	range (usage)	range (metamodel)	invalid values	mode	default (metamodel)	default(API/tool)
1	EAttribute.lowerBound	[-2069046262, 2000]	[0, ∞]	-1,-2069046262	0	NA	0
2	EAttribute.upperBound	[-1427576723, 2147483647]	[-2, ∞]	-3,-1427576723	1	1	1
3	EReference.lowerBound	[-1826932651, 384]	[0, ∞]	-1,-1826932651	0	NA	0
4	EReference.upperBound	[-2, 2147483647]	[-2, ∞]		1	1	1
5	EOperation.lowerBound	[-1253265258, 99]	[0, ∞]	-1253265258	0	NA	0
6	EOperation.upperBound	[-795180990, 95]	[-2, ∞]		1	1	1
7	EParameter.lowerBound	[-1827820016, 99]	[0, ∞]	-1827820016	0	NA	0
8	EParameter.upperBound	[-1, 222661208]	[-2, ∞]		1	1	1

The ranges indicate what values attributes can have based on the metamodel, and what they actually have in our dataset. Invalid values are the actual attribute values we encounter in our dataset, which are outside the theoretical range

Table 13 Statistics and information on an illustrative set of string attribute values

	attribute	mode	significant top 2-5	default (metamodel)	default (API/tool)
1	EPackage.nsURI	null	http://www.eclipse.org/emf/2002/Ecore, http://www.eclipse.org/acceleo/anydsl, http://www.bank.com, http://UML2.ecore	NA	NA
2	EPackage.nsPrefix	null	cim, ecore, model, uml	NA	NA
3	EDataType. instanceClassName	org.eclipse.emf.common.util.Enumerator	java.lang.String , null, java.lang.Object, float	NA	NA, not on GUI
4	EDataType. instanceTypeName	org.eclipse.emf.common.util.Enumerator	java.lang.String , null, java.lang.Object, float	NA	NA
5	EEnum. instanceClassName	null	–	NA	NA, not on GUI
6	EEnum. instanceTypeName	null	–	NA	NA
7	EClass. instanceClassName	null	java.util.Map$Entry	NA	NA, not on GUI
8	EClass. instanceTypeName	null	java.util.Map$Entry	NA	NA
9	EAttribute. defaultValueLiteral	null	false, true, 0, 1	NA	NA
10	EReference. defaultValueLiteral	null	–	NA	NA

We list the most frequent values including the mode, and other significant top 2-5 values in our dataset

8.3.1 Least Used or Misused Attribute Values

Results For all the boolean attributes, both values are used, i.e. there is no case where a true or false value is completely ignored. In some cases, the values are used in a balanced manner, while in others the percentages are in overwhelming favor for a single value. We have identified 10 cases where a single value was used $> 95\%$ of the time, while in 3 of those the usage is extremely one-sided: *EAttribute.iD*, *EReference.unique* and *EEnum.serializable* with close to 0-1 percent of usage for one value (two of which are shown in Table 11).

There are only a few integer values, representing the lower and upper cardinality bounds for the four concrete subclasses of *ETypedElement*. Table 12 shows that indeed a wide range of values are used for the attributes, hence mostly legitimizing the choice of integer domain. Nevertheless, it is worthwhile noting that for an overwhelming majority of the cases ($> 99.9\%$, omitted in the Table for simplicity) the *lowerBound* values are 0 or 1, and the *upperBound* values 1, -1 or -2. Note that EMF explicitly allows the particular negative values -1 and -2, to represent unbounded (-1) and unspecified (-2)(Steinberg et al. 2008). There are a few invalid values which we encountered in a small number of metamodels, which are shown in Table 12.

We have identified a number of string or other valued attributes besides the model element names (e.g. *EAttribute.name*), which we could include in our analysis. We skip the attributes related to *EAnnotation*, as we handle those separately in the following section. Following Table 13, we can see for some cases with a null or non-null mode, together with a significant list of top frequent items, e.g. for *EPackage.nsURI/nsPrefix* and *EDataType.instanceClassName/instanceTypeName*. For *EENum.instanceClassName/ instanceTypeName* and *EReference.defaultValueLiteral*, however, we observed an extremely high ($> 99\%$) frequency of the null value and no other significantly frequent value. Finally, we have observed a peculiarity in the use of *instanceClassName* vs *instanceTypeName*. *instanceTypeName*, while not mentioned in the EMF book (Steinberg et al. 2008), is a part of the Ecore meta-metamodel. *instanceClassName*, according to the EMF API, “represents the actual Java instance class that this meta-object represents” while *instanceTypeName* “represents the parameterized Java type that this meta-object represents.” We verify this by checking a simple random sample of the cases (size 50) in our dataset. In the majority of the cases we checked, they are the same. For a small number of cases, however, *instanceTypeName* represents a more concrete or parametrized version of the type in *instanceClassName*. Examples would include *instanceTypeName* being `java.lang.Iterable`, while *instanceTypeName* for the same EClass set as `java.lang.Iterable<EObject>` and even `Iterable<? extends ...IFormalParameter>` in another case.

Discussion Some boolean attributes, such as *unique*, are a part of the abstract metaclass *ETypedElement* and inherited in e.g. *EAttribute* and *EReference*. While for some submetaclasses this inheritance makes sense, for others it may not. Indeed, the EMF reference book (Steinberg et al. 2008) states: “Two other attributes of ETypedElement are meaningful only in the multiplicity-many case: *unique* specifies whether a single value is prevented from occurring more than once, and ordered specifies whether the order of values is significant. Currently, their use is limited. The behavior of a multi-valued attribute depends on its uniqueness, but references always behave as if unique is true.” This explains why it is barely used in EReference (item 8 on Table 11). In our results, we observe that it is even rarer for *EOperation* and *EParameter* (both $\geq 99\%$ true, see items 10 and 11 on Table 11). A similar argument holds for *serializable*. For the concrete metaclass *EDataType*, the usage of *serializable* can be justified with the values as 93% true and 7% false. However, for the

EEnum metaclass as a subclass of *EDataType*, it is almost never set to false. We conclude from our results, almost all *EEnums* are regarded to be serializable by the users. EMF states that "the difference between these two approaches is seen at development time: if a data type is serializable, a generated factory will include methods to convert values of that type to and from strings" (Steinberg et al. 2008). Finally, *EAttribute.id* is intended to determine whether the attribute can uniquely identify the instance of the containing class. It seems this is rarely used in practice.

Next we discuss the integer attributes. In practice, the metamodels have different semantic domains, which correspond to a wide range of possible values. The choice of integers (in contrast to, for instance, limited ranges with enumerations) is well justified. Due to the natural range of integers as $(-\infty,\infty)$, however, we observed invalid values not intended by the language design: Negative values for *lowerBound*, and negative values beyond -2 for *upperBound*.

Among the string attributes, for *instanceClassName* and *instanceTypeName*, we observe a similar pattern as above for the attribute *serializable* for *EDataType* and *EEnum*. For the *EDataType* metaclass, the two attributes are used with a variety of values. This makes sense as these represent the actual external (e.g. Java) correspondents for the *EDataTypes*. This is not the case for the sub-metaclass *EEnum* however: it is rarely associated with an external instance class or type. A similar pattern holds for the attribute *defaultValueLiteral*: it is defined in the abstract metaclass *EStructuralFeature*. While being used in the *EAttribute* sub-metaclass with a range of values, it is almost exclusively null for *EReferences*, where it does not make sense to have a default value derived from a string literal. For the situation with *instanceClassName* vs. *instanceTypeName*, it is also worthwhile to note that there seems to be simple tool support in Eclipse EMF for entering *instanceTypeName* through the GUI, while none for *instanceClassName*.

Suggestions For the boolean attributes, we suggest removing *EAttribute.iD* and moving *ETypedElement.unique* to the sub-metaclass *EAttribute*, where it is actually used and makes sense. Another design change could be making *EDataType* abstract (e.g. *EAbstractDataType*) with two concrete instantiations: *EDataType* and *EEnum*. Putting the *serializable* attribute under the new concrete *EDataType* would avoid its redundancy in the *EEnum* metaclass instances. As already mentioned before, the architects have opted for using the integer type with occasional negative values to represent unbounded (with *upperBound* set to -1) or undefined (with *upperBound* set to -2) multiplicities for certain elements. While adding new custom built-in data types could be a way to go, a more lightweight alternative is limiting the possible values through imposing constraints and relying on the EMF Validation Framework (see Chapter 18 in the EMF book (Steinberg et al. 2008)). This is already the case at the moment, therefore we do not have further suggestions on these issues. Finally, we discuss the string attributes. Similarly to the cases for the integer attributes of *unique* and *serializable* attribute, we suggest moving *EStructuralFeature.defaultValueLiteral* to the sub-metaclass *EAttribute*, and changing the metaclass hierarchy so that *instanceClassName* and *instanceTypeName* are only part of the concrete *EDataType* and not *EEnum*. A further simplification would be making *instanceClassName* derived (i.e. the non-parametric version) from *instanceTypeName* to avoid redundancy and inconsistency.

8.3.2 Default Values vs. Mode Values

Results We distinguish the actionable cases where the default value (1) exists and differs from the mode, (2) exists on the tool/API level but not on the metamodel level, or (3) does not exist at all while having a significantly frequent mode value. We have identified only *EReference.containment* for case (1). For case (3), there are the metaclasses *EAttribute, EReference, EOperation, EParameter* having the attribute *required* (with $> 65\%$ false), with no default on the metamodel, and not even shown on the GUI. We classify the remaining attributes (i.e. the majority in the metamodel) as case (2).

Discussion Default values have a lot to do with the actual usage of a language, and only be properly set after conducting a large-scale empirical analysis. At design time, the architects have their own goals and motivations to set these, which might not correspond 100% with how users actually use the language.

Suggestions We suggest setting default values to modes on the language level (as opposed to just the tool or API level) for all the boolean and integer valued attributes across cases (1), (2) and (3). For the string valued attributes, the mode value is either null or a not-so-frequent one (e.g. org.eclipse.emf.common.util.E- numerator for *EDataType.instanceClassName* as shown in Table 13). For those, we do not suggest setting a default value.

8.3.3 Most Often Used Values

Results We have not found any meaningful case where only a handful of values ($<$ 10 as considered by Herrmannsdoerfer et al. (2010)) are used for e.g. an integer or string valued attribute. The only case falling into this category is the following: *EReference.defaultValueLiteral* (8 unique values) with an overwhelming percentage of the value null, hence suggested being removed above.

Discussion The choice of the attribute types seems proper for the language, therefore we do not suggest any change in the attribute types.

8.4 Specific Parts of the Language

In this section, we study the specific parts of the language, namely annotations and generics, in closer detail. Our motivation for focusing on these parts will be elaborated in each subsection.

8.4.1 Annotations

As discussed above, *EAnnotation* metaclass and its contents (notably *EStringToStringMapEntry*) are among the most used model elements in our dataset. Therefore, below we discuss the usage of EAnnotation in more detail.

Research Questions We identify the following research questions to be investigated in this subsection.

- **RQ4.1** What are the often used types of *EAnnotations*, as indicated by their *source* attribute?
- **RQ4.2** What *EStringToStringMapEntry* keys are frequently used with each other and the corresponding *EAnnotation* types?
- **RQ4.3** For those frequently co-occurring sets of keys, are there also frequently used values?

Table 14 Most frequent ($> 1\%$) values for *EAnnotation.source*

	source	count	percentage
1	http://www.eclipse.org/emf/2002/genmodel	711,682	0.39
2	http://org/eclipse/emf/ecore/util/extendedmetadata	590,201	0.32
3	of_generation	82,908	0.05
4	subsets	53,605	0.03
5	http://iec.ch/tc57/2009/cim-schema-cim14	39,930	0.02
6	http://www.eclipse.org/uml2/1.1.0/genmodel	26,487	0.01
7	redefines	19,150	0.01
8	hidden	15,398	0.01
9	asstring	15,245	0.01
10	http://www.eclipse.org/emf/2002/ecore	13,938	0.01
11	taggedvalues	12,588	0.01
12	http://langdale.com.au/2005/uml	12,587	0.01
13	http://iec.ch/tc57/2010/cim-schema-cim15	11,594	0.01
14	extendedmetadata	11,399	0.01
15	http://www.eclipse.org/uml2/2.0.0/uml	11,006	0.01
16	metadata	10,407	0.01
17	http://www.polarsys.org/capella/mnoe/capellalike/mapping	9,456	0.01

Often used types of EAnnotations. As seen in Table 14, we have mixed cases of the standard (as suggested by the language designers (Steinberg et al. 2008)) and custom usage of annotations. Items 1, 2 and 10 are the standard ways of describing consecutively code generation, non-Ecore details, and general-purpose model information. Items 6 and 15 correspond to similar elements for Eclipse UML[30]. There are a number of for specific modeling languages: 5, 12 and 13 for CIMTool[31], and 17 for Capella[32]. The rest are used for custom extensions. Comparing these with the top 5 values reported by Herrmannsdoerfer et al. (2010), we see that the two most frequent values are the same. On the other hand, we can see the remaining three values reported by them, i.e. *taggedvalues, metadata and subsets* in Table 14 with lower percentages. One item, namely *stereotype in their top-6 list does not appear in our list. We attribute this to more 'in-the-wild' nature of our dataset, while their dataset comprises Eclipse repositories as a big majority.* Note that, despite a relatively high percentage of usage overall in our findings, these annotation sources are typically used in small sets of (ranging from a few to a few hundreds) metamodels for a single tool, originating from a handful of repositories/users, etc.; they do not represent global conventions. Another observation is that the remaining standard annotations as suggested by the EMF book (Steinberg et al. 2008) and literature (e.g. on OCL) are rarely used: XSD2Ecore, EMOF tags and OCL constraints. This phenomenon can be further investigated in a dedicated follow-up study.

Often co-occurring EStringToStringMapEntry keys We have mined the association rules (using the R package *arules*[33]) for *EAnnotation* types and the keys they contain in *EString-*

30 https://www.eclipse.org/modeling/mdt/?project=uml2

31 https://wiki.cimtool.org/

32 https://www.eclipse.org/capella/

33 https://github.com/mhahsler/arules

Table 15 Frequently co-occurring keys along with source types (last element in each item list) within *EAnnotations*

	LHS	RHS	support	confidence	lift
1	{source:emf-genmodel}	{documentation}	0.41	0.98	2.16
2	{source:extendedmetadata}	{name}	0.34	0.98	2.77
3	{source:extendedmetadata}	{kind}	0.32	0.91	2.83
4	{kind,source:extendedmetadata}	{name}	0.32	1.00	2.83
5	{name,source:extendedmetadata}	{kind}	0.32	0.93	2.89
6	{namespace,source:extendedmetadata}	{kind}	0.18	0.98	3.08
7	{namespace,source:extendedmetadata}	{name}	0.18	0.98	2.80
8	{kind,namespace,source:extendedmetadata}	{name}	0.18	1.00	2.83
9	{name,namespace,source:extendedmetadata}	{kind}	0.18	1.00	3.12
10	{group,source:extendedmetadata}	{kind}	0.03	0.97	3.04
11	{group,source:extendedmetadata}	{name}	0.03	0.97	2.75
12	{group,kind,source:extendedmetadata}	{name}	0.03	0.99	2.82
13	{group,name,source:extendedmetadata}	{kind}	0.03	1.00	3.12
14	{group,source:extendedmetadata}	{namespace}	0.03	0.92	4.92
15	{group,namespace,source:extendedmetadata}	{kind}	0.03	1.00	3.12
16	{group,kind,source:extendedmetadata}	{namespace}	0.03	0.94	5.05
17	{group,namespace,source:extendedmetadata}	{name}	0.03	0.99	2.82
18	{group,name,source:extendedmetadata}	{namespace}	0.03	0.94	5.04
19	{group,kind,namespace,source:extendedmetadata}	{name}	0.03	0.99	2.82
20	{group,name,namespace,source:extendedmetadata}	{kind}	0.03	1.00	3.12
21	{group,kind,name,source:extendedmetadata}	{namespace}	0.03	0.94	5.04
22	{source:uml2-genmodel}	{body}	0.02	1.00	31.82
23	{basetype,source:extendedmetadata}	{name}	0.01	0.83	2.36

Note that long URLs for sources have been shortened to appear as emf-genmodel, uml2-genmodel and extendedmetadata

ToStringMapEntry instances. The association rules with support ≥ 0.01 (i.e. found at least 1% of the instances in our dataset) and lift > 1 (i.e. significantly co-occurring beyond conditional independence) are given in Table 15. Following from the previous paragraph on the different EAnnotation types, we only focus on standard *EAnnotations* for this part, and have excluded the non-standard ones from Table 15. We can see that typically (support 0.41) the *documentation* key is used with GenModel (emf-genmodel) annotations. Only for the UML2 variant of GenModel (uml2-genmodel), we occasionally (support 0.02) see a significantly (lift 31.82) co-occurring *body*. The rest of the official suggestions (e.g. *suppressedGetVisibility*) are not very often used. As for the Extended Metadata (extendedmetadata) annotations, we observe a frequent (support 0.34, 0.32) use of *name, kind*, and further occasional co-occurrence of *group, namespace, basetype*. We can trace the use of such keys in the EMF book for extending EMF: for instance, using *kind* to specify complex types, or *basetype* to restrict data types (Steinberg et al. 2008).

Often co-occurring source and key-value pairs We have investigated the top 50 most frequent sources and key-value pairs for *EAnnotation*, for the standard types, as mentioned above. We have not found any rules, which are (a) for standard source types, (b) frequent enough (i.e. support ≥ 0.01 and (c) are relevant for lifting as a language construct or predefined suggestions as a part of tool support. We have manually inspected two rules satisfying (a) and (b), and discarded as irrelevant due to them being documentation text too specifically related to caching and diagnostics.

Suggestions For the standard types of annotations ExtendedMetaData and GenModel (as described in the EMF reference book (Steinberg et al. 2008)), there is already tool support, in the sense that a user can directly create those types of EAnnotations with the source automatically set accordingly in the Eclipse EMF editor. Creating specialized language constructs would be redundant for those cases. Instead, we suggest improving tool support so that when the user chooses to create the special annotation types through the GUI, the most frequently used keys are automatically created and attached as *EStringToStringMapEntry* items (i.e. through *EAnnotation.details*). We cannot suggest any particular values for the keys given the wide range of possible string values in the usage range, as we could not find any meaningful association rules corresponding to values. They can be left empty, to be filled by the user. Our verdict from this exercise of performing association rule mining on the key-value pairs is that, there are no immediately useful patterns or insights in the scope of our study.

8.4.2 Generics

While we have studied the use of generics in the previous sections and concluded that they are not used very often, we would like to dig deeper and find out more details on their usage. The EMF reference book (Steinberg et al. 2008) specifies several variants of generic types: simple generic (using simple type arguments), parameterized and wildcard. Generic and non-generic types manifest themselves as the type of *ETypedElements*, as well as generic supertypes of *EClass* and generic exceptions to *EOperation*.

Research Questions We identify the following research questions to be investigated in this subsection.

– **RQ5.1** What percentages of *EClassifier* and *EOperation* instances contain type parameters?
– **RQ5.2** Which specific types of generics are used overall?
– **RQ5.3** How often do *EClass* instances use generic supertypes and *EOperation* instances generic exceptions?

Results We present the statistics on type parameters in Table 16. We can see that, though rare overall, type parameters are still used in *EClass*, *EDataType* and *EOperation* instances. They are almost never used for *EEnum*, which is not unexpected since type parameters do not make much sense for enumerations in the metamodeling context. The specific types of generics used are in turn presented in Table 17. An overwhelming majority (96%) of all concrete *ETypedElement* instances (i.e. *EAttribute*, *EReference*, *EOperation* and *EParameter*) have non-generic types, which is not surprising given the generics study in the previous sections. However, the results reveal that the majority of instances are used in the form of parameterized generic types, followed by some simple generic types and very rarely wildcard types. Finally, the Tables 18 and 19 show the statistics on the use of generics for supertypes and exceptions. While there is a somewhat occasional use of generic supertypes, they are virtually no generic exceptions.

Table 16 Type parameter count and percentages per metaclasses

	metaclass	# with type parameters	# all	percentage
1	EClass	2,686	850,237	0.00315
2	EDataType	1,601	54,349	0.02943
3	EEnum	7	61,867	0.00011
4	EOperation	1,371	236,957	0.00579

Table 17 The non-generic and generic types used in our dataset, with counts and percentages

	types	count	percentage
1	non-generic	2,405,298	0.96199
2	simple generic	3,859	0.00154
3	parameterized generic	90,971	0.03638
4	wildcard generic	218	0.00009

Table 18 The non-generic and generic supertypes used in our dataset, with counts and percentages

	supertypes	count	percentage
1	non-generic	688,862	0.99216
2	generic	5,441	0.00784

Table 19 The non-generic and generic exceptions used in our dataset, with counts and percentages

	exceptions	count	percentage
1	non-generic	4,281	0.99953
2	generic	2	0.00047

Suggestions As previously explained, the generics feature is implemented as an overhaul of the complete type system; even non-generic types imply the use of *EGenericType* instances in a specific way. Therefore, it is not entirely feasible to suggest small changes with little impact on the consistency of the metamodel. Given the extreme rare usage and for the sake of simplicity for the language, however, one could think of removing/disallowing the following (e.g. through validation rules if not possible to change the metamodel): use of generic types for *EEnum*, wildcard type generics in general, and generic exceptions for *EOperation*. Note that an evolution perspective could potentially reveal more insights into the use of generics, since this feature was introduced not at the beginning of EMF, but rather along its evolution. Since evolution is out of scope for this study, we leave it for future work to investigate.

9 Threats to Validity

Our study is subject to several threats to validity with the classification scheme of Wohlin et al. (2012), which are elaborated in this section.

Conclusion Validity Conclusion validity is concerned with the relation between the analysis and the outcome, with regards the conclusions being reasonable. For the statistical analyses, we mitigated this by not assuming any normality or symmetry in our data, therefore reporting e.g. median values next to the mean values, and reporting confidence and lift values for the association rules we mined. Beyond these points, the statistical techniques we have performed are very basic and well-established ones, therefore not likely to pose any threat themselves to the conclusions we derive about the usage of EMF metamodels. However, the interpretation of the usage how and why's, as well as the suggestions on the language improvements are subjective. This threat to conclusion validity is mitigated in several ways. The first author is an experienced researcher with a PhD degree and 10+ years of experience with EMF metamodeling and language design. We rigorously consulted the EMF reference book for the theory, while manually inspected the data where necessary (e.g. outliers) to get a better insight before coming to any conclusions. A methodologically sound interview with the language architect of EMF would contribute greatly to this study; however, it is not possible due to resource constraints by the potential interviewees. Final remarks include (1) our study can only argue based on what is present in the metamodels, and cannot reach to any conclusion about what is missing and should be added to the language, and (2) since we do not investigate the actual models belonging to the metamodels, we do not paint a precise picture on metamodel usage within the whole ecosystem.

Internal Validity Internal validity indicates whether the results really do follow from the data and not any external or uncontrolled factors. Given the nature of our study (i.e. observational), we do not claim causality in our quantitative analyses. However, in discussing e.g. why a certain language feature is not used by the practitioners in our dataset, there can be external factors which we cannot account for. We believe this can best be mitigated by a follow-up study of a different nature, e.g. a controlled experiment.

Construct Validity Construct validity is concerned with the experiment design and the degree to which it accurately reflects (e.g. measures) the underlying theoretical construct. We have utilized a relatively large workflow for this study which realizes a set of theoretical steps ranging from data collection and filtering to feature extraction and analysis steps. The tools we have used in the workflow might pose threats to construct validity. PHANTOM, which we used for filtering out non-engineered projects for instance, is validated on GitHub projects in general and works on a project-basis. It might not be suited for detecting engineered metamodeling repositories as well as filtering out individual non-engineered metamodels within engineered repositories. The deduplication protocol also employs some design decisions, such as including repositories with presumably reused and repurposed in different contexts. This might in some cases lead to overrepresentation of some metamodels, hence the language constructs used in them. Furthermore, we had to exclude 1k+ models due to them being unprocessable. While these represent a small portion of the whole dataset, they might lead to several advanced use cases being omitted from our analysis. SAMOS, on the other hand, is a tool for model analytics in general and relies on certain simplifications and small errors leading to a not 100% accurate representation of metamodels in the Neo4j database. We have performed several steps of manual quality assurance (e.g. by comparing a set of random models with their representations in Neo4j) to minimize this threat. Finally, our analysis involves multiple steps over the span of several weeks, and GitHub continuously evolves in the meantime. As a result, we have observed minor yet unavoidable inconsistencies (e.g. repositories being deleted) in between the steps, an example of which is the different number of repositories reported in Tables 2 and 3.

External Validity External validity is concerned with the generalization of the conclusions beyond the context of our study. We have employed a data collection phase (see Section 4) with multiple sources (GitHub and other repositories) and acquisition methods (GitHub REST API, GHTorrent etc.) to achieve a large coverage of the target population, as all the (engineered) metamodels on GitHub (master/default branches only). On top of the large coverage, we have performed a best-effort filtering and deduplication. While there are a number of false positives in these steps, these are the state-of-the-practice automated techniques used in the literature. Therefore, we believe our dataset adequately represents the target population of EMF metamodels in public engineered repositories on GitHub, and our results generalize to the target population. The additional effort we have put to investigate open source repositories other than GitHub has revealed relatively few other metamodels in open source. Nevertheless, we do not claim any generalization of our findings to open source in general or e.g. EMF metamodeling in industry.

10 Related Work

In this section, we briefly discuss the related work. One influential recent work involves mining of UML models from GitHub (Hebig et al. 2016; López et al. 2021), and studying the practices and perceptions of their usage (Ho-Quang et al. 2017). The authors perform a language usage analysis for UML models, along with different analyses around the software development process and evolution, such as their introduction and modification across the project lifespan. In another recent study, Heinze et al. mine business process (BPMN) models from GitHub, yet focus on quality analysis rather than language usage (Heinze et al. 2020b). Regarding DSLs, Tairas and Cabot conducted a language usage and clone analysis study on Puppet and Atlas Transformation Language, both of which are textual DSLs (Tairas and Cabot 2015). Another very relevant work by Lämmel and Pek studies the usage of P3P, a domain-specific language for privacy policies. They present an extensive analysis of vocabulary usage, correctness, syntactic and semantic metrics, cloning and language extensions. Object Constraint Language (OCL) has also been the focus of a few language usage studies (Cadavid et al. 2015; Mengerink et al. 2019). While the OCL language works on top of EMF and is strongly related to EMF metamodels, we do not discuss those studies in detail in this paper due to our focus on the metamodels themselves.

More directly related work on EMF metamodels includes an early study by Herrmansdoerfer et al. on metamodel usage, including EMF metamodels as well as in other languages (Herrmannsdoerfer et al. 2010). Since we share common objectives with this study while having a larger and more recent dataset (the EMF part), we provide an extensive discussion and comparison with their work where applicable in Section 8. However it is worthwhile to emphasize that the scope of our study is larger than (the EMF part of) Herrmansdoerfer et al., both in the volume of data and analysis themes. As for the dataset, we performed a state-of-the-art MSR study with rigorous steps, notably data collection and deduplication. We claim our dataset is not only a more recent one, but also a methodologically sound one in terms of representing the EMF metamodels in open source. Still, it is interesting to see that, in the part where we can compare our results with theirs in detail (i.e. metaclass usage), the results are very similar.

An additional dataset of EMF metamodels has been collected from public Eclipse repositories by Kögel and Tichy (2018): we discuss this dataset in Section 4. There we concluded that most Eclipse repositories are mirrored on GitHub, therefore included in our study focusing on GitHub. It is worthwhile to add that Kögel and Tichy's dataset also includes version history of the artifacts, and might be very useful when studying the evolution of EMF metamodels over time. Williams et al. (2013) present preliminary metrics and evolution analysis for 500+ EMF metamodels collected from various sources including GitHub, Google Code and internal project repositories. Di Rocco et al. (2014) measure metrics and their correlation for 450+ metamodels. Both Williams et al. and Di Rocco et al. tackle a much smaller dataset and perform a smaller set of analyses, while some overlap with our study such as metaclass counts and use of abstract metaclasses. Finally, there are studies on EMF usage in open source which focus on the ecosystem, technology model, architecture and process aspects, rather than language usage for metamodels (Di Rocco et al. 2020; Härtel et al. 2018; Heinz et al. 2020; Kolovos et al. 2015). In those articles, the scope of the artifacts under study is larger than ours; including, for instance, code generation or model transformation linked to the metamodels. A major advantage of such approaches is that they can include semantics and pragmatics in their study, while in our case we are limited to syntactic analysis.

Last but not least, there is a line of research focusing on the use of EMF in industry rather than open source. EMF is indeed reported to be an industry standard (Steinberg et al. 2008), and there are empirical studies in the literature. Though, most related work focuses on a bigger picture beyond just EMF (e.g. the use of MDE, its benefits and so on (Hutchinson et al. 2011; Mohagheghi and Dehlen 2008)). A fine-grained study of the artifacts themselves is relatively rare to the best of our knowledge, arguably due to limited availability and confidentiality of data. Notable exceptions include our previous work where we compare the usage of OCL in industry and open source (Mengerink et al. 2017) and present metrics from EMF technology stack in industry (Babur et al. 2017; Mengerink et al. 2017).

11 Conclusion and Future Work

In this paper, we have presented an extensive empirical study into the language usage of EMF metamodels on GitHub. Motivating the need for such a study, e.g. for the sake of language improvement and taming, we have employed a workflow consisting of the following steps: (1) data collection from GitHub via various methods, (2) filtering out non-engineered and duplicated projects, (3) extracting model and repository data to be represented in a graph database, and finally (4) performing queries post-processing for obtaining the analysis results on language usage. We have conducted various analysis on metaclass, attribute, feature/relationship usage as well as specific parts of the language: annotations and generics. We present the statistical results accompanied by a set of actionable suggestions for language improvement. To name a few of the potential language improvements, we suggest modifications to metaclasses (e.g. making them abstract), relationships (e.g. removing unused ones), attribute types and default values on the level of the language as well as tool support. By having such a feedback loop, the language design can be not only better tailored to meet the expectations of its user base, but also improved with respect to certain qualities (e.g. simplicity, restriction) which might help the language architect in the first place (e.g. with testing). We believe this study fills a gap in the realm of domain-specific languages and model-driven engineering and will hopefully help future development of DSL/MLs, whether it be EMF directly or other languages, as an exemplary work from a methodological point of view.

This paper lays the foundation for further research by ourselves and other researchers. First of all, the meticulously collected and curated dataset itself is valuable for further empirical studies. One such study is an even more in-depth analysis of the language usage, with closer look into other details regarding the language (e.g. dependencies and referencing among metamodels), different sub-populations (e.g. basic and advanced usage of EMF, metamodels for forward vs. reverse engineering) as well as the whole EMF ecosystem especially with the models conforming to the metamodels (cf. Härtel et al. (2018); Heinz et al. (2020)). A more thorough elaboration of the results and further suggestions can be made in light of the existing literature (Jácome and De Lara 2018), but also notably working in tandem with the language designers. Other empirical studies based on the dataset that we plan to do in future include extensive analysis on metamodel quality and technical debt, such as fine-grained cloning and modeling errors. While the current dataset is limited to a static view of the repositories (i.e. no commit history for the metamodels) and very limited metadata (only owner and repository information with basic timestamps for creation and last update), we plan to enhance the dataset and do further empirical analyses on evolution, process and social aspects of metamodeling. This would enable, for instance, a longitudinal analysis of modeling errors. A further direction for future work would be to comparatively analyze the design and usage of other languages (e.g. MOF, UML and other metamodels as analyzed by Herrmannsdoerfer et al. (2010)), see what is missing in the language and suggest actionable improvements.

Declarations

Conflicts of interest The authors have no conflicts of interest to declare that are relevant to the content of this article.

References

Allamanis M (2019) The adverse effects of code duplication in machine learning models of code. In Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, p 143–153

Andova S, van den Brand MGJ, Engelen LJP, Verhoeff T (2012) MDE basics with a DSL focus. In International School on Formal Methods for the Design of Computer, Communication and Software Systems, p 21–57. Springer

Babur Ö (2019) Model analytics and management. PhD thesis, Technische Universiteit Eindhoven. Proefschrift

Babur Ö, Cleophas L (2017) Using n-grams for the automated clustering of structural models. In International Conference on Current Trends in Theory and Practice of Informatics, p 510–524. Springer

Babur Ö, Cleophas L, van den Brand M (2016) Hierarchical clustering of metamodels for comparative analysis and visualization. In European Conference on Modelling Foundations and Applications, p 3–18. Springer

Babur Ö, Cleophas L, van den Brand M (2019) Metamodel clone detection with SAMOS. Journal of Computer Languages 51:57–74

Babur Ö, Cleophas L, van den Brand M (2022) SAMOS - a framework for model analytics and management. Sci Comput Program 223:102877

Babur Ö, Cleophas L, van den Brand M, Tekinerdogan B, Aksit M (2017) Models, more models, and then a lot more. In Federation of International Conferences on Software Technologies: Applications and Foundations, p 129–135. Springer

Baltes S, Ralph P (2020) Sampling in software engineering research: A critical review and guidelines. arXiv preprint. arXiv:2002.07764

Basciani F, Rocco JD, Ruscio DD, Iovino L, Pierantonio A (2016) Automated clustering of metamodel repositories. In Advanced Information Systems Engineering: 28th International Conference, CAiSE 2016, Ljubljana, Slovenia, June 13-17, 2016. Proceedings vol 28. Springer, pp 342–358

Biber D, Douglas B, Conrad S, Reppen R (1998) Corpus linguistics: Investigating language structure and use. Cambridge University Press

Brambilla M, Cabot J, Wimmer M (2017) Model-driven software engineering in practice, second edition. Synthesis Lectures on Software Engineering 3(1):1–207

Broy M, Kirstan S, Krcmar H, Schätz B (2012) What is the benefit of a model-based design of embedded software systems in the car industry? In Emerging Technologies for the Evolution and Maintenance of Software Models, p 343–369. IGI Global

Cadavid JJ, Combemale B, Baudry B (2015) An analysis of metamodeling practices for MOF and OCL. Comput Lang Syst Struct 41:42–65

Clark T, Van den Brand M, Combemale B, Rumpe B (2015) Conceptual model of the globalization for domain-specific languages. In Globalizing Domain-Specific Languages, p 7–20. Springer

Combemale B, France R, Jézéquel J-M, Rumpe B, Steel J, Vojtisek D (2016) Engineering modeling languages: Turning domain knowledge into tools. CRC Press

Concas G, Marchesi M, Pinna S, Serra N (2007) Power-laws in a large object-oriented software system. IEEE Trans Softw Eng 33(10):687–708

Cosentino V, Izquierdo JLC, Cabot J (2017) A systematic mapping study of software development with GitHub. IEEE Access 5:7173–7192

Cosentino V, Izquierdo JLC, Cabot J (2016) Findings from GitHub: methods, datasets and limitations. In 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR), p 137–141. IEEE

de F. Farias MA, Novais R, Júnior MC, da Silva Carvalho LP, Mendonça M, Spínola RO (2016) A systematic mapping study on mining software repositories. In Proceedings of the 31st Annual ACM Symposium on Applied Computing, p 1472–1479

de Mello RM, Stolee KT, Travassos GH (2015) Investigating samples representativeness for an online experiment in java code search. In 2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), p 1–10

Di Rocco J, Di Ruscio D, Härtel J, Iovino L, Lämmel R, Pierantonio A (2020) Understanding mde projects: megamodels to the rescue for architecture recovery. Softw Syst Model 19:401–423

Di Rocco J, Di Ruscio D, Iovino L, Pierantonio A (2014) Mining metrics for understanding metamodel characteristics. In Proceedings of the 6th International Workshop on Modeling in Software Engineering, p 55–60

Erdweg S, Van Der Storm T, Völter M, Boersma M, Bosman R, Cook WR, Gerritsen A, Hulshout A, Kelly S, Loh A et al (2013) The state of the art in language workbenches. In International Conference on Software Language Engineering, p 197–217. Springer

Favre J-M, Gasevic D, Lämmel R, Pek E (2010) Empirical language analysis in software linguistics. In International Conference on Software Language Engineering, p 316–326. Springer

Gabriel P, GoulÃ?Â£o M, Amaral V (2010) Do software languages engineers evaluate their languages? In Franch JPCX, Gimenes I (eds) XIII Congreso Iberoamericano en, p 149–162. CIbSE2010, 04

Gharehyazie M, Ray B, Keshani M, Zavosht MS, Heydarnoori A, Filkov V (2019) Cross-project code clones in GitHub. Empir Softw Eng 24(3):1538–1573

Gousios G, Spinellis D (2012) GHTorrent: GitHub's data from a firehose. In 2012 9th IEEE Working Conference on Mining Software Repositories (MSR), p 12–21. IEEE

Grechanik M, McMillan C, DeFerrari L, Comi M, Crespi S, Poshyvanyk D, Fu C, Xie Q, Ghezzi C (2010) An empirical investigation into a large-scale java open source code repository. In Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, p 1–10

Härtel J, Heinz M, Lämmel R (2018) Emf patterns of usage on github. In European Conference on Modelling Foundations and Applications, p 216–234. Springer

Hebig R, Quang TH, Chaudron MRV, Robles G, Fernandez MA (2016) The quest for open source projects that use UML: mining GitHub. In Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, p 173–183

Heinz M, Härtel J, Lämmel R (2020) Reproducible construction of interconnected technology models for emf code generation. J Object Technol 19(2):8–1

Heinze TS, Stefanko V, Amme W (2020) Mining BPMN processes on GitHub for tool validation and development. In: Nurcan S, Reinhartz-Berger I, Soffer P, Zdravkovic J (eds) Enterprise, Business-Process and Information Systems Modeling. Springer International Publishing, Cham, pp 193–208

Herrmannsdoerfer M, Ratiu D, Koegel M (2010) Metamodel usage analysis for identifying metamodel improvements. In International Conference on Software Language Engineering, p 62–81. Springer

Ho-Quang T, Hebig R, Robles G, Chaudron MRV, Fernandez MA (2017) Practices and perceptions of UML use in open source projects. In 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), p 203–212. IEEE

Hutchinson J, Whittle J, Rouncefield M, Kristoffersen S (2011) Empirical assessment of mde in industry. In Proceedings of the 33rd international conference on software engineering, p 471–480

Information technology - Meta Object Facility (MOF) (2005) Standard, International Organization for Standardization

Izquierdo JLC, Cosentino V, Cabot J (2017) An empirical study on the maturity of the eclipse modeling ecosystem. In 2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS), p 292–302. IEEE

Jácome S, De Lara J (2018) Controlling meta-model extensibility in model-driven engineering. IEEE Access 6:19923–19939

Kalliamvakou E, Gousios G, Blincoe K, Singer L, German DM, Damian D (2014) The promises and perils of mining GitHub. In Proceedings of the 11th working conference on mining software repositories, p 92–101

Kalliamvakou E, Gousios G, Blincoe K, Singer L, German DM, Damian D (2016) An in-depth study of the promises and perils of mining GitHub. Empir Softw Eng 21(5):2035–2071

Kögel S, Tichy M (2018) A dataset of EMF models from eclipse projects

Kolovos DS, Matragkas ND, Korkontzelos I, Ananiadou S, Paige RF (2015) Assessing the use of eclipse MDE technologies in open-source software projects. In OSS4MDE@ MoDELS, p 20–29

Kolovos DS, Rose LM, Matragkas N, Paige RF, Guerra E, Cuadrado JS, Lara JD, Ráth I, Varró D, Tisi M et al (2013) A research roadmap towards achieving scalability in model driven engineering. In Proceedings of the Workshop on Scalability in Model Driven Engineering, p 1–10

Lämmel R, Pek E (2013) Understanding privacy policies: A study in empirical analysis of language usage. Empir Softw Eng 18:310–374

Lopes CV, Maj P, Martins P, Saini V, Yang D, Zitny J, Sajnani H, Vitek J (2017) Déjàvu: a map of code duplicates on GitHub. Proceedings of the ACM on Programming Languages 1(OOPSLA):1–28

López JAH, Izquierdo JLC, Cuadrado JS (2021) Modelset: a dataset for machine learning in model-driven engineering. Softw Syst Model, p 1–20

Manning CD, Raghavan P, Schütze H et al (2008) Introduction to information retrieval 1. Cambridge University Press

Melton H, Tempero E (2007) An empirical study of cycles among classes in java. Empir Softw Eng 12(4):389–415

Mengerink J, Noten J, Schiffelers R, van den Brand M, Serebrenik A (2017) A case of industrial vs. open-source ocl: not so different after all. In ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS 2017), p 472–474. CEUR-WS. org

Mengerink JGM, Noten J, Serebrenik A (2019) Empowering ocl research: a large-scale corpus of open-source data from github. Empir Softw Eng 24(3):1574–1609

Mengerink JGM, Serebrenik A, Schiffelers RRH, van den Brand MGJ (2017) Automated analyses of model-driven artifacts: obtaining insights into industrial application of mde. In Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement, p 116–121

Mohagheghi P, Dehlen V (2008) Where is the proof?-a review of experiences from applying mde in industry. In Model Driven Architecture–Foundations and Applications: 4th European Conference, ECMDA-FA 2008, Berlin, Germany, June 9-13, 2008. Proceedings vol 4, pp 432–443. Springer

Mohamed MA, Challenger M, Kardas G (2020) Applications of model-driven engineering in cyber-physical systems: a systematic mapping study. Journal of Computer Languages 59:100972

Muller MJ, Kuhn S (1993) Participatory design. Commun ACM 36(6):24–28

Munaiah N, Kroh S, Cabrey C, Nagappan M (2017) Curating GitHub for engineered software projects. Empir Softw Eng 22(6):3219–3253

Nagappan M, Zimmermann T, Bird C (2013) Diversity in software engineering research. In: Meyer B, Baresi L, Mezini M (eds) Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18–26, 2013. ACM, pp 466–476

Noten J, Mengerink JGM, Serebrenik A (2017) A data set of OCL expressions on GitHub. In 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), p 531–534. IEEE

Pagán JE, Cuadrado JS, Molina JG (2011) Morsa: A scalable approach for persisting and accessing large models. In International Conference on Model Driven Engineering Languages and Systems, p 77–92. Springer

Paige RF, Ostroff JS, Brooke PJ (2000) Principles for modeling language design. Inf Softw Technol 42(10):665–675

Pickerill P, Jungen HJ, Ochodek M, Staron M (2020) PHANTOM: Curating GitHub for engineered software projects using time-series clustering. Empir Software Eng

Pietri A, Spinellis D, Zacchiroli S (2019) The software heritage graph dataset: public software development under one roof. In 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), p 138–142. IEEE

Qiu D, Li B, Barr ET, Su Z (2017) Understanding the syntactic rule usage in java. J Syst Softw 123:160–172

Ray B, Posnett D, Filkov V, Devanbu P (2014) A large scale study of programming languages and code quality in GitHub. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, p 155–165

Ruta D, Gabrys B (2005) Classifier selection for majority voting. Information fusion 6(1):63–81

Spinellis D, Kotti Z, Mockus A (2020) A dataset for GitHub repository deduplication. arXiv preprint. arXiv:2002.02314

Steinberg D, Budinsky F, Paternostro M, Merks E (2008) EMF: Eclipse Modeling Framework Second Edition. Pearson Education

Stol K-J, Fitzgerald B (2018) The abc of software engineering research. ACM Trans Softw Eng Methodol (TOSEM) 27(3):1–51

Tairas R, Cabot J (2015) Corpus-based analysis of domain-specific languages. Softw Syst Model 14(2):889–904

Tekinerdogan B, Babur Ö, Cleophas L, van den Brand M, Akşit M (2019) Introduction to model management and analytics. In Model Management and Analytics for Large Scale Systems, p 3–11. Academic Press

Wieringa RJ (2014) Design science methodology for information systems and software engineering. Springer

Williams JR, Zolotas A, Matragkas ND, Rose LM, Kolovos DS, Paige RF, Polack FAC (2013) What do metamodels really look like? Eessmod@ Models 1078:55–60

Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A (2012) Experimentation in software engineering. Springer Science & Business Media

Authors and Affiliations

Önder Babur[1,2] · Eleni Constantinou[2,3] · Alexander Serebrenik[2]

Eleni Constantinou
constantinou.a.eleni@ucy.ac.cy

Alexander Serebrenik
a.serebrenik@tue.nl

1 Information Technology Group, Wageningen University & Research, Wageningen, The Netherlands

2 Department of Mathematics and Computer Science, Eindhoven University of Technology, Eindhoven, The Netherlands

3 Department of Computer Science, University of Cyprus, Nicosia, Cyprus