Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Reproducible Construction of Interconnected Technology Models for EMF Code Generation

Marcel Heinz[a] Johannes Härtel[a] Ralf Lämmel[a]

a. Software Languages Team, University of Koblenz-Landau, Germany

Abstract Software technologies such as the Eclipse Modeling Framework (*EMF*) involve complex usage scenarios that need to be understood or communicated by newcomers, developers, teachers, contributors, and others. Such different stakeholders consult scattered resources that provide *textual explanations* and *code examples* that cover different facets of a technology. More specifically, textual explanations can be located in i) developer literature that describes idiomatic usage of a technology, and in ii) scientific literature that explains technology from a formal and abstract perspective. Code examples can be located in iii) demo projects that exemplify intended usage, and in iv) wild projects that provide complex code examples in actual applications. In this paper, we propose annotated megamodels of technology usage as macroscopic summaries, i.e., *technology models*; these models interconnect the scattered textual explanations and code examples. We present a methodology for the incremental construction of technology models in a reproducible manner. It relies on the systematic reduction of a corpus to 'useful' evidence for each increment. The manual effort of searching for representative links is reduced by dedicated queries. We exemplify the application of the methodology for technology models that summarize *EMF* code generation.

1 Introduction

Motivation – How to understand *EMF*? The Eclipse Modeling Framework (*EMF*) is a technology which is tightly integrated into the Eclipse IDE and serves as a solution for basic MDE application scenarios [SBMP08]. To use *EMF*, several aspects need to be understood: metamodels, models, code units, workflows, and translations may affect an *EMF*-based language to work properly; dedicated points (also in generated code) can be used for the customization of an *EMF*-based language's model and editor; one needs to use two or more running Eclipse instances; each instance is configured with different plugins; the developer has to deal with three orthogonal resource layers (operating system, Eclipse workspace, and *EMF* resource system); several design patterns are involved (Adapter, Command, Registry, and Proxy).

Marcel Heinz, Johannes Härtel, Ralf Lämmel. *Reproducible Construction of Interconnected Technology Models for EMF Code Generation*. In Journal of Object Technology, vol. 19, no. 2, 2020, pages 8:1–25. doi:10.5381/jot.2020.19.2.a8

Resources for Understanding Software Technologies Textual explanations and code examples have been identified as essential resources for understanding API (method) usage [Rob09, RLP13, RC15]. We assume that the problem of identifying such resources naturally transfers to understanding technology usage beyond technologies other than APIs. The use of technology may involve artifacts other than programs – notably models in the case of *EMF*. We consider four types of resources: i.) *Developer literature* provides informal *textual explanations*. In the search for textual explanations, we focus on literature that most likely covers all facets of idiomatic technology usage. In the case of *EMF*, the content of the book written by Steinberg et al. [SBMP08] provides such coverage. (Tutorials and forum posts can be consolidated as well, but their structure and quality challenge systematic exploitation.) ii.) *Scientific literature* tends to provide more abstract *textual explanations*. In the case of *EMF*, papers either explain how *EMF* works [BBC+05, HGG12] or how it may be adapted and integrated [EIG+15, GGKdL14, LWWC12]. iii.) *Demo projects* are typically linked in *developer literature* to provide a collection of referential *code examples* which suggest idiomatic usage. iv.) *Wild projects*, such as open source projects on GitHub, provide more complex *code examples* that represent actual technology usage and allow insights on how the technology is commonly used.

Contribution – Reproducible, Interconnected Technology Models We propose to interconnect textual explanations and code examples through megamodels of technology usage, i.e., technology models. In our research on software technology, we have modeled which and how technologies are used in specific software projects through megamodels [FLV12, LSV13, LV14a, HLV17, HHL+17, SLH+17]. In this paper, we focus on the reproducibility of technology models through existing resources. To this end, we formulate the following research question:

> *How can we construct a technology model in a reproducible manner so that it is interconnected with textual explanations and code examples?*

We answer the research question essentially by means of presenting a validated methodology that involves formulating and executing queries to reduce the manual effort of searching for textual explanations and code examples in selected corpora. In this context, a *query* is any pattern or algorithm that leads to the reduction of the search space. Importantly, to assure reproducibility, the queries and the resulting links to textual explanations and code examples are persisted.

Evaluation Based on *EMF* Code Generation While the methodology has been generally inspired by our previous work, we evaluate it specifically in this paper in the context of *EMF* code generation by executing the methodology directly for two exemplary technology models and provide a detailed discussion. The first technology model summarizes idiomatic usage and the second demonstrates how a misconception in a technology model is revealed and hence prevented in the construction process. All artifacts that have been created for the evaluation by examples are available online.[1]

Road-map of the Paper Section 2 presents an illustrative technology model of *EMF* code generation. Section 3 introduces the methodology. Section 4 applies the methodology to *EMF* code generation. Section 5 explains limitations. Section 6 summarizes related work. Section 7 concludes the paper.

[1] `https://github.com/softlang/megaemf`

2 *EMF* Code Generation

Different resources exist, where *EMF* usage is explained. In non-scientific literature, we find text and loose visual diagrams[2] that explain *EMF* code generation. In MDE literature, many researchers have summarized code generation using *EMF* in different contexts, such as developer activities [HGG12], adapted *EMF* processes [EIG+15] and pluggable analysis [HHL+17]. *EMF* has served as an exemplary technology several times in our research [FLV12, HLV17, HHL+17, SLH+17, HHL18].

In this paper, we are concerned with the aspect that, within the context of education or for the purpose of useful documentation of a software technology, any technology model needs to be reproducible so that it can be safely reused and referred to. Therefore, we develop a methodology for the reproducible construction of technology models. We execute this methodology specifically to construct a reproducible technology model on *EMF* code generation.

The technology model depicted in Figure 1 serves as the running example. It provides a visual summary of the central artifact types and their relations that are often covered in non-scientific as well as scientific literature. It relates to five types of artifacts that are instantiated when using *EMF*. It summarizes how three different types of Java code artifacts are derived using an Ecore and generator model. In megamodeling, such derivations have been modeled as functions and their applications [HLV17, HHL+17, Zay12, FLV12, LV14b].

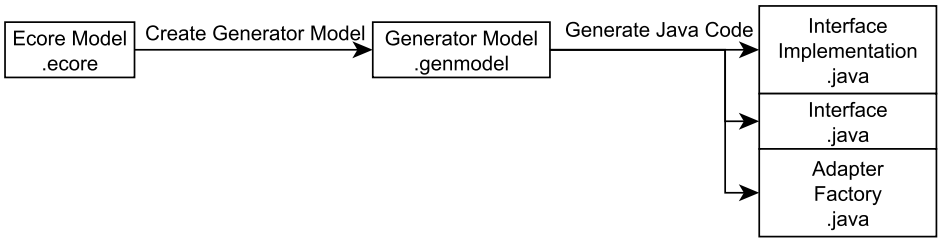

Figure 1 – A technology model of *EMF* code generation.

3 Methodology

We propose an incremental process to construct reproducible technology models. Technology models consist of technology-specific artifact types and their relations. In the process, artifact types and relations are added as increments one after another. For every increment to the model, *evidence* is needed. Hence, every increment needs to be aligned with concise textual explanations and idiomatic code examples. By linking the *evidence*, the construction process becomes reproducible. *Linked* textual explanations and code examples add value to a "meaningless diagram".

(Query-based) Reduction Steps: Searching for textual explanations and code examples in a *corpus* requires manual effort. Figure 2 summarizes the iterative procedure to systematically reduce a *corpus* of resources to *evidence* that is then *linked* to the technology model. If *evidence* is already known from personal *experience*, it is *linked* immediately. As long as *evidence* is missing, the goal is to reduce the *corpus*

[2]See, for example, `https://eclipsesource.com/blogs/tutorials/emf-tutorial/`

by executing several manual as well as automated steps. At first, a promising *corpus* that contains *evidence* needs to be selected. Then, a *query* is developed. Queries are *formulated* and *executed* to reduce the search scope within a selected *corpus*. Here, a query is any tool-based reduction of the corpus to candidates for *linked* evidence, for example, by searching for an artifact type's name using *grep*. *Queries* can be formulated based on pattern identified in previously *linked evidence* and returned *query results*. The *query results* are then manually inspected to confirm concise textual explanations and idiomatic code examples and link them as *evidence*. The detection of *evidence* is a continuous process. Textual explanations can be helpful to identify code examples and vice versa. Hence, we do not intend to enforce any order in which the different corpora are processed. They can be processed in an interleaving manner.

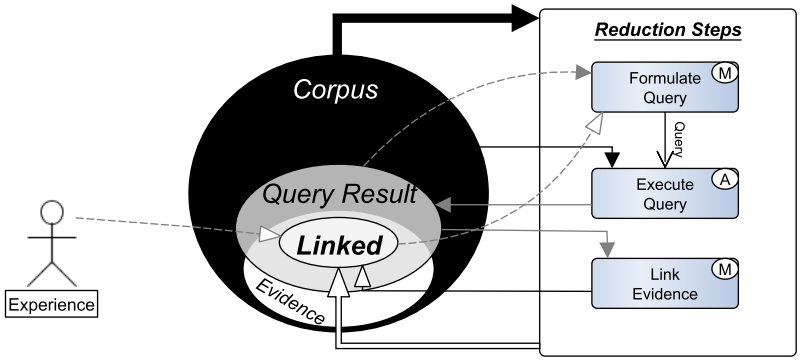

Figure 2 – Manually ('M') or automatically ('A') executed steps reduce a corpus to linked evidence. Resources are related to steps by input and output edges, whose color hints at whether it is unknown (black), query-related (gray), or linked (white).

The degree of manual effort for executing the methodology depends on the given *experience*. If concise textual explanations and idiomatic code examples can be linked without any querying effort, the effort is at the minimum. If no query can be formulated from the beginning, the effort is at the maximum. Not every resource that can serve as evidence for an increment may be returned by a developed query, especially, within a restricted time window; and not every query result can serve as evidence. We focus on what is in between: Concise textual explanations and idiomatic code examples that are selected from systematically refined query results and then linked. To assure the reproducibility of evidence, queries are shared as well. In the example-driven evaluation, we emphasize reproducibility by providing *reduction step protocols* in which we record input and output of each executed reduction step. This way, a reduction step protocol instantiates the reduction procedure from Figure 2.

When the construction of a technology model is based on an unvaried or non-representative corpus of resources, it is prone to errors, in particular, misconception (See Section 4.2). Linking *evidence* helps in raising the confidence in an interconnected technology model. By developing *queries* varied and representative *evidence* can be linked. Sharing the *queries* makes the recovery of *evidence* reproducible and facilitates linking evidence in any additional corpus. Thus, we assume that systematically interconnecting a varied and representative set of resources is more robust and may help with constructing an interconnected technology model more quickly, when compared to a less systematic approach.

3.1 Stakeholders

We discuss five different stakeholders that benefit from an interconnected technology model or the methodology.

Newcomers (as in EMF newcomers in our case) benefit from the result of the modeling effort – linked textual explanations and code examples; the model may help with understanding intertwined complex aspects [BY06]. That is, newcomers are not at all assumed to execute the methodology, because they miss the *experience* that is required for selecting *evidence*.

Developers (as in developers with some EMF knowledge in our case) benefit from interconnected technology models to revise their knowledge; they may not be interested in creating one; they can validate their own mental model against a given technology model; they can find additional contextual insights from linked code examples and textual explanations; they typically execute analogous steps already, because the steps are natural to gain understanding of complex technology, e.g., by searching and inspecting code samples or querying documentation [Rob09].

Teachers (as in teachers wanting to cover EMF in our case) execute the methodology and communicate their findings. This group also includes code reviewers [BB13], when they need to argue on what is the idiomatic ("correct") usage of a complex framework, such as *EMF*. They can provide the technology model itself; they can communicate the linked textual explanations and code examples to illustrate their *experience*; they can also create new code examples that demonstrate technology usage according to textual explanations. There is also the related stakeholder of *authors* (as in authors on content describing or involving EMF in our case) who may want to describe the technology in a systematic and structured and comprehensive manner; authors would benefit from executing the methodology, as they usually perform similar steps.

Contributors (as in contributors to EMF itself in our case) execute the methodology and communicate their hands on *experience*. They mix the properties of teachers and developers. Most notably, this group can most reliably execute the methodology to identify precise textual explanations and idiomatic code examples.

3.2 Textual Explanations

A common measure to communicate the trustworthiness of information is to link high-qualitative literature as *evidence*. Here, we discuss the use of scientific and non-scientific literature. We refer to the latter as developer literature in order to emphasize the main audience. Overall, the goal is to link concise textual explanations as *evidence* in a selected *corpus*.

3.2.1 Developer Literature

Books and web documentation typically serve as lexicons of code examples, which are accompanied by dedicated textual explanations. They assist developers at understanding idiomatic usage. In the past, we have identified and processed developer literature as text-based corpora on different technologies, e.g., in [SLH+17]. Furthermore, we are inspired by the search for textual explanations on API types based on

natural language pattern [PRM15] and also consider relation extraction, e.g., based on distant supervision [MBSJ09] as a potential approach to querying. The reduction is subject to steps as follows:

- **Formulate Query:** Either, the name of an artifact type or relation is the query. Or, parts or synonyms of a name can be concatenated to form a query. Any pattern-based algorithms, e.g., based on Hearst Pattern [Hea92], uses the names, parts, or synonyms as words that are in a grammatical relationship.
- **Execute Query:** By executing a query, we reduce a text-based corpus, e.g., a whole book, to promising text passages that may potentially serve as *evidence*. Basic tools for executing a query are *grep* and *CTRL+F*. Advanced tools are NLP-pipeline implementations. At first, structural elements can be used as the input instead of the whole text. For example, a glossary already refers to pages that introduce specific concepts; a title of a chapter or (sub-) section hints at central topics; a visual diagram emphasizes the importance of concepts.Then, the query is applied to text passages within the respective scope, e.g., the text in a specific subsection.
- **Link Evidence:** *Query results* are reduced to textual explanations, which actually provide *evidence*. A textual explanation potentially encompasses multiple text passages and visual diagrams from which knowledge can be inferred. Such complex textual explanations can be left out whenever more concise textual explanations can be linked. A concise explanation is always preferable over an ambiguous text.

3.2.2 Scientific Literature

Technologies are typically covered by research papers in that the papers explain technology usage in the context of a research contribution from a more formal perspective. With respect to scientific sources, the goal is to reduce a corpus of papers, for example, Google Scholar, to papers that provide concise textual explanations. For scientific literature, the reduction steps can be compared to a systematic literature survey [KBB+09]. Thus:

- **Formulate Query:** A query is a search string. It is formulated in analogy to developer literature. The names of multiple artifact types and relations can be used in a single search string to identify a paper.
- **Execute Query:** By executing a query, we reduce a corpus of papers to candidates, where artifact types or relations are mentioned. The formulated query is either executed by using a search engine, e.g., Google Scholar, or by processing downloaded PDF files. A reasonable subset of papers can be taken from the first results returned by a search engine. This set of query results can be steadily increased by considering additional results or by adapting the formulated query.
- **Link Evidence:** At last, papers are linked that provide concise explanations. Linking one paper can be enough, but multiple papers can complement each other by the diversity of points of views and contexts. Research papers can cover more advanced technology usage based on modifications such as an integration or adaption [EIG+15]. Papers that explain adapted technology usage are not useful for teaching idiomatic technology usage. Thus, they are not considered.

3.3 Code Examples

The exhaustive recovery of code examples is motivated in our previously conducted analysis of technology usage [HHL18]. The corpus, subject to code reduction, is a set of software projects that relate to part of the technology model. Queries are formulated and executed to reduce code to potential code examples.

3.3.1 Demo Projects

Developer literature is often accompanied by demo projects, which are typically simple to understand compared to arbitrary 'wild' projects. Developing an accurate query for code examples is non-trivial. The textual explanations from developer literature can add to the necessary *experience* on how to recognize code examples. Formulating a query that correctly recognizes all code examples requires systematic debugging and constant inspection of intermediate query results. Hence, the goal is to correctly reduce the corpus of demo projects to all code examples in them for the technology model. This effort is feasible, because it facilitates the reduction of a more complex corpus, i.e., large wild projects, to code examples. We suggest these steps:

- **Formulate Query:** A query is formulated based on known indications. File extensions and keywords can be used to discover code that relates to artifact types. References can be resolved to find potential code examples of a relation.
- **Execute Query:** By executing a query, all artifacts in selected software projects are reduced to potential code examples. Feasible tools to execute queries may range from rule-based reasoners [HHL18] to handcrafted miners, e.g., *grep*.
- **Link Evidence:** For demo projects, no reduction takes place when linking *evidence*. The results from the queries overlap with the actual code examples. In demo projects, all code examples are assumed to be idiomatic.

3.3.2 Wild Projects

Other than demo project, software projects in the 'wild' are not systematically focusing on exemplifying technology usage. Such projects reflect the actual usage of a technology. Different kinds of repositories exist on GitHub ranging from projects developed by students to professionally maintained open source frameworks, like ANTLR or Xtext. GitHub can be queried by using its API to identify a set of promising repositories, where the technology is used [HHL18, KMK+15]. Here, the goal is to reduce a corpus of wild projects to code examples that tend to be more complex than those in demo projects. We suggest these steps:

- **Formulate Query:** The queries that have been developed for demo projects can be reused. While the queries have been tested against sandbox examples, complex wild projects may introduce additional difficulties. In this case, query results need to be manually sampled and inspected. Then, patterns need to be determined within the sample to systematically improve the queries. Such sampling is illustrated in Section 4.1. If the queries are adapted, the linked code examples from demo projects and wild projects are used for regression tests. This way, we make sure that queries are improved and not changed for the worse.
- **Execute Query:** Executing a query is analog to demo projects.

- **Link Evidence:** The selection of useful code examples in wild projects depends on who the interconnected technology model is shared with. Complex logic and mixed use of other technologies are obstacles to correctly understanding idiomatic technology usage. While considering all code examples can be interesting to gain empirical insights, a small set of handpicked code examples can be sufficient for teaching.

4 Evaluation by Examples

We evaluate the methodology by executing and discussing it in detail for two different technology models. First, we demonstrate a smooth construction process for a technology model of *EMF* code generation, where all parts of the technology model can be linked to evidence in every type of resource. Second, we discuss how executing the methodology helps to reveal misconceptions in a technology model.

The methodology can be executed with any set of tools. For illustration, we cover different tools to execute queries in literature as well as software projects. To query for textual explanations, we demonstrate the use of *CTRL+F*. To query for code examples, we present two tools. For the smooth construction, we use the query engine that is developed in [HHL18]. For the misconception, we recover code examples by using the GitHub search API.

4.1 Modeling Code Generation

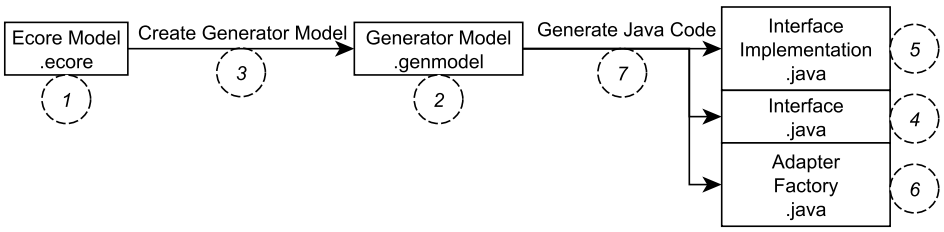

Figure 3 – We incrementally construct the technology model from Figure 1 in the order that is depicted here (1-7). We present the respective evidence in this Section.

The technology model in Figure 1 is inspired by a conceptual model in [HGG12]. It summarizes code generation in *EMF* through the following increments that were added in this order: every *EMF* project contains an *Ecore model*, which is recognizable by a '.ecore' file extension; a *generator model* encodes the configuration for code generation; the *generator model* is derived from an *Ecore model* (see *Create Generator Model*); we consider three types of generated Java files, namely an *interface*, an *interface implementation*, and an *Adapter Factory*; all generated Java files are derived from the *generator model* (see *Generate Java Code*).

4.1.1 Developer Literature

For textual explanations, we focus on a corpus that most likely covers all facets of idiomatic technology usage. For *EMF*, the *corpus* is the content of the book written by Steinberg et al. [SBMP08]. It covers all facets of *EMF* usage in detail. It can be seen as the referential corpus, because it is cited in most of the related scientific

literature and it is advertised in a banner ad on EMF's website that says: 'Buy the book. Hence, it is recommended by *contributors* of *EMF*.

In this exemplary construction process, our goal is to link initial textual explanations that define artifact types and their relations. Instead of processing all text passages, structural elements, such as a table of content, can be used to reduce the effort. For each artifact type and relation, we identify relevant subsections by running queries against the table of contents. Figure 4 presents exemplary titles of (sub-) sections in [SBMP08]. The section titles help us to identify primary textual explanations of artifact types and relations. For the type *Generator model*, a relevant subsection can be easily recognized. The artifact type is initially defined in Subsection 2.4.4. 'The Generator Model'.

2.4 Generating Code 23
2.4.1 *Generated Model Classes* 24
2.4.2 *Other Generated "Stuff"* 26
2.4.3 *Regeneration and Merge* 27
2.4.4 *The Generator Model* 28

Figure 4 – The section titles help us to identify subsections, where types and relations are initially defined, for example, the artifact type *Generator model* is defined in the respective Subsection 2.4.4. 'The Generator Model'.

An excerpt of a *reduction step protocol* is sketched in Table 1, where each row documents one executed reduction step. In the presented examples, we name the respective input and output of each executed reduction step to identify the textual explanation for `Ecore model`. The query only relates to one word ('Ecore') of the increment's name ('Ecore model'). The query is run against the table of contents. Thus, it returns the titles subsections.

ID	***Step***	***In***	***Out***	***Automation***
1	Formulate Query	*Experience*: Name 'Ecore model'	*Query*: 'Ecore'	M
2	Execute Query	*Corpus Resource*: Table of Content *Query*: See **1**	*Query Results*: Subsection 2.3.1, Subsection 2.3.5 Subsection 4.2.4 ...	A
3	Link Evidence	*Query results:* See **2**	*Linked*: Subsection 2.3.1	M
...	...	...	...	...

Table 1 – Excerpt of the reduction step protocol for the developer literature, specifically, for the type *Ecore model*.

The links to textual explanations are accompanied by dedicated rationales in Table 2. Explanations of `Ecore model` are introduced in Subsection 2.3.1; in the same query results, we also identify Subsection 12.4.4 that is called "Ecore2GenModel" which explains the relation `Create Generator model`; introductory explanations on the generator model can be found by querying for "Generator" in Subsection 2.4.4; in the query results for "Generator", we also identify Subsection 12.4.5 that is called "Generator", which provides the textual explanations for the relation `Generate Java Code`. Java artifacts and hints at the relation `Generate Java Code` can be identified in Section 2.4.1 by querying for "Generate". In the initially linked subsections, there is no concise explanation on the derivation of the generator model or code genera-

tion. Concise explanations are presented later in Subsection 12.4.4 Ecore2GenModel and Subsection 12.4.5 Generator, where the processes that trigger the derivation are explicitly named and explained.

Type/**R**elation	Links	Rationale
T:Ecore Model	Subsection 2.3.1 "The Ecore (Meta) Model"	- Explains what an Ecore model is and what its structural parts represent.
T:Generator Model	Subsection 2.4.4 "Generator Model"	- Explains that a generator model contains configuration aspects.
R:Create Generator Model	Subsection 2.4.4 "Generator Model"	- Explains that the generator model refers to and decorates the Ecore model with code generation config.
	Subsection 12.4.4 "Ecore2GenModel"	- Explains the process artifact which creates the generator model based on an Ecore model.
T:Interface	Subsection 2.4.1 "Generated Model Classes"	- Explains that an interface is generated and is a subtype of EObject.
T:Interface Implementation	Subsection 2.4.1 "Generated Model Classes"	- Explains that a respective interface implementation is also generated.
T:Adapter Factory	Subsection 2.4.2 "Other Generated "Stuff""	- Explains what the skeleton Adapter Factory is used for.
R:Generate Java Code	Subsection 2.4.1 "Generated Model Classes"	- Explains that Java interfaces as well as their implementations are generated.
	Subsection 2.4.2 "Other Generated Stuff"	- Explains that an adapter factory is generated, e.g., POAdapterFactory.
	Subsection 12.4.5 "Generator"	- Explains the process artifact which executes code generation.

Table 2 – Types and relations are linked to textual explanations in the book. The quoted text passages and rationales support the links to relevant subsections.

4.1.2 Scientific Literature

We find scientific literature evidences for a technology model by querying Google Scholar. In Google Scholar, query results are already ranked according to relevance based on: 'the full text of each document, where it was published, who it was written by, as well as how often and how recently it has been cited in other scholarly literature.'[3] Table 3 presents reduction step protocol. First, we combine words from the names of all artifact types and relations to a single query in row *1.* Only ten results are returned. We generally skip books and slide decks as well as student's work, such as M.Sc. or diploma theses. For the naive query (see row 1, Table 3), we only find the paper written by Hebig [HGG12] that confirms all artifact types and relations as expected, because our technology model is inspired by a contained diagram. Then, we formulate a shorter search string: `'EMF "generator model" generate Java'`. 429 results are returned. We process papers in the order of their appearance in query

[3] `https://scholar.google.com/intl/en/scholar/about.html`

results until we have linked five results that redundantly cover artifact types and relations.

ID	*Step*	*In*	*Out*	*Automation*
1	Formulate Query	*Experience*: All names of types	*Query*: 'EMF "Ecore model" "Generator model" Interface Implementation "Adapter Factory"'	M
2	Execute Query	*Corpus Resource*: Google Scholar *Query*: See **1**	*Query Results*: 2x *EMF* book 2x eclipsecon slides 5x Student theses Hebig et al [HGG12]	A
3	Link Evidence	*Query results:* See **2**	*Linked*: Hebig et al [HGG12]	M
4	Formulate Query	*Expertise*: Focus on 'Java code is derived from a generator model'	*Query*: EMF "generator model" generate Java	M
5	Execute Query	*Corpus Resource*: Google Scholar	*Query Results*: 429 heterogeneous results	A
6	Link Evidence	*Query results:* See **5**	*Linked*: [KRbA$^+$10, BEJ10], [BS15, KGRP17]	M

Table 3 – The reduction step protocol for the scientific literature.

Table 4 summarizes the results of processing scientific literature. We execute the queries that we used for the developer literature and execute them with *CTRL+F* in the reduced set of papers (see row 3 and 6, column "Out" in Table 3). By reading the returned text paragraphs that contain a query result, we are able to determine which artifact type and relation is covered. In the end, we either simply advise to read the paper, or we highlight the existence of central figures explicitly. The rationales explain why a highlighted figure is linked and also cite central textual explanations when we link a paper as a whole.

Type/**R**elation	*Links*	*Rationale*
All	[HGG12, Fig. 3]	Uses the same names.
T:Ecore model, T:Generator model, R:Create Generator Model	[KRbA$^+$10, Fig. 1]	Text and model refer to: "Ecore metamodel", "GenModel model", "EMF Ecore2GenModel transformation".
T:Ecore model, T:Generator model, R:Create Generator Model	[BEJ10]	The introduction explains the dependency of the generator model to Ecore model on the fragment level.
All except -T:Adapter Factory	[BS15]	'The EMF code generator is always invoked on a so called generator model'
All except -T:Adapter Factory	[KGRP17]	'the GenModel is consumed by a built-in model-to-text transformation'

Table 4 – Linked scientific literature that explains the artifact types and relations.

4.1.3 Demo Project

The *EMF* book [SBMP08] is accompanied by a set of projects that can be downloaded online[4]. They are demo projects. We select one demo project as our *corpus*. It is called 'PrimerPO' and illustrates the basics described up to Chapter 4. If we wanted to focus on advanced *EMF* usage, such as the use of different code pattern, we would have to systematically process Chapter 10, construct technology models that represent the pattern, and then search for code examples in all the demo projects. For now, this basic demo project is sufficient.

We use a rule-based approach (*QegaL*) for the automated detection of code examples in GitHub repositories [HHL18]. Inference of type and relation instances is driven by monotonously adding facts to a triple store. Thus, artifacts are represented by URIs and relations are encoded as triples. During the inference, declarative rules search in the triple store and a project for new facts. If one of the rules succeeds, it adds the respective triple(s) – adding a new triple triggers all rules again. This process stops, when no facts can be added anymore. Table 5 summarizes executed reduction steps for the demo projects, which are fully committed to the query engine to perform an exhaustive analysis on GitHub.

ID	***Step***	***In***	***Out***	***Automation***
1	Formulate Query	*Experience*: search for file endings .java, .genmodel, .ecore	*Query*: see Listing 1 + '.java' query	M
2	Execute Query	*Corpus Resource*: Project PrimerPO *Query*: See **1**	*Query Results*: PrimerPO.ecore PrimerPO.genmodel Item.java .. PPOPackage.java ...	A
3	Link Evidence	*Query results:* See **2**	*Linked*: see Table 6	M
...	...	...	...	...

Table 5 – Excerpt of the reduction step protocol for the demo project PrimerPO. '.java' files returned by the query are manually filtered. For instance, *PPOPackage* does not exemplify any modeled type.

In an initial query, we follow the file-ending information that is given in Figure 3. Listing 1 presents the documented queries that are implemented in *QegaL*.

```
(?ecoreModel, sl:manifestsAs, sl:File) //every file
Extension(?ecoreModel,"ecore") //with the file extension '.ecore'
→ (?ecoreModel, sl:instanceOf, sl:EcoreModel). //is an Ecore model

(?generatorModel, sl:manifestsAs, sl:File) //every file
Extension(?generatorModel,"genmodel") //with the file extension '.genmodel'
→ (?generatorModel, sl:instanceOf, sl:GeneratorModel). //is a Generator model
```

Listing 1 – *QegaL* queries identify examples of the Ecore- and generator model.

We detect the derivation of the generator model from an Ecore model by identifying an encoded reference. For instance, the file 'PrimerPO.genmodel' contains the XMI element `<foreignModel>PrimerPO.ecore</foreignModel>`. The rules for detecting the derivation relation based on this tag and the common parent folder are presented and documented in Listing 2.

[4] `http://www.informit.com/store/emf-eclipse-modeling-framework-9780321331885`

```
(?generatorModel, sl:instanceOf, sl:GeneratorModel) //Generator Model
(?generatorModel, sl:partOf, ?folder) //Folder
StrXml2(?generatorModel,"//foreignModel/text",?foreignModel) //Ecore name via XPATH
UriConcat(?folder,"/",?foreignModel, ?ecoreModel) //folder + Ecore name = Ecore URI
→ (?ecoreModel, sl:CreateGeneratorModel,?generatorModel).
```

Listing 2 – *QegaL* queries to identify code examples for the derivation relation between the Generator model and the Ecore model.

Developing accurate queries is complicated. Hence, we use links to code examples that should not result from a query as test artifacts. Such test artifacts can also be shared. The principle is exemplified in Listing 3. We interpret the artifact type `Interface` as an interface corresponding to a model class. Here, '*Package.java' and '*Factory.java' files do not exemplify interfaces. The file 'PpoPackage.java' is also an interface but not a model class. Next, it is assured that the adapter factory is not confused with the file 'PpoFactory.java'.

```
<:/src/ppo/PpoPackage.java> sl:instanceOf sl:Interface.
<:/src/ppo/PpoFactory.java> sl:instanceOf sl:Interface.
<:/src/ppo/PpoFactory.java> sl:instanceOf sl:AdapterFactory.
```

Listing 3 – We test based on false-positive triples.

To abbreviate explanations, Figure 5 summarizes the ideas behind the queries to recognize code examples for the relation `Generate Java Code`. Previous research can be consulted for more details on incremental querying [HHL18]. Chapter 4 of the *EMF* book can be consolidated for the required *experience* to develop the query.

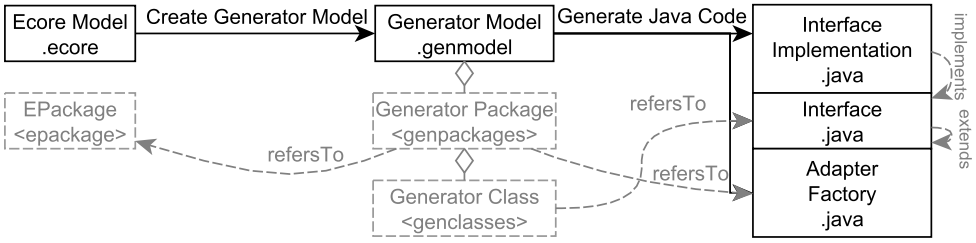

Figure 5 – This sketch illustrates the ideas for querying for code examples of the technology model. The type hierarchy and references in the generator model are resolved.

The type hierarchy and references from generator models to Java files need to be resolved for accurate queries. We use the following indications for code examples. Ecore models and generator models contain packages and classes in analogy to the generated code. For every generator package, there exists a Java package. For every generator class, there exists a Java interface and an implementation. To identify code examples of the respective Java interfaces, we search for interfaces that (transitively) extend 'EObject'. Then, a respective interface implementation can be found based on its 'implements' reference in the class signature that refers to a previously identified interface. As the last artifact type, adapter factories can be recovered based on an 'extends' reference in the signature of an interface to the interface 'AdapterFactory'. Next, we recover code examples of the relation `Generate Java Code` itself. We identify each generator package in the generator model. Each generator package refers to a respective Ecore package. The Ecore package has to be part of an Ecore model that we have identified earlier. We build qualified names, such as 'ppo.Item', to identify each generator class in the generator model. The qualified name of the Java package, where the generated code is located, can be derived from the respective Ecore

package, because the namespace is defined there. For every Java file, we determine the qualified names as well. In the end, we match the qualified names of generator classes with the qualified names of interfaces. Interface implementations can then be connected with generator classes based on the respective interfaces. Through the generator classes, the generated Java code is connected with the generator model.

Because we need to assure that all code examples are matched correctly, we manually persist links to the correct code examples in the demo project first. Table 6 presents respective exemplary links to code examples of artifact types and relations.

Type/**R**elation	*Links*	*Rationale*
T:EM	`<:/model/PrimerPO.ecore>`	Ecore Model Query (see Listing 1)
T:GM	`<:/model/PrimerPO.genmodel>`	Generator Model Query (see Listing 1)
R:EtoG	(`<:/model/PrimerPO.ecore>`, `<:/model/PrimerPO.genmodel>`)	Foreign Model Query (see Listing 2)
T:Int	`<:/src/ppo/Item.java>,..`	Extends Queries (see Figure 5)
T:Impl	`<:/src/ppo/impl/ItemImpl.java>,..`	Implements Queries (see Figure 5)
T:AF	`<:/src/ppo/util/PpoAdapterFactory.java>`	Package Reference Query (see Figure 5)
R:GtoJ	(`<:/model/PrimerPO.genmodel>`, `<:/src/ppo/Item.java>`), .. , (`<:/model/PrimerPO.genmodel>`, `<:/src/ppo/impl/ItemImpl.java>`), .. , (`<:/model/PrimerPO.genmodel>`, `<:/src/ppo/util/PpoAdapterFactory.java>`)	Reference Queries (see Figure 5)

Table 6 – Linked code examples from the demo project *PrimerPO*. We only link one code example for each artifact type. More links can be viewed online. Abbreviations: EM=Ecore Model, GM=Generator Model, EtoG=Create Generator Model, Int=Interface, Impl=Interface Implementation, AF=Adapter Factory, GtoJ=Generate Java Code.

4.1.4 Wild Project

Based on an investigation in wild repositories, we gain insights into whether the technology model in Figure 1 is commonly instantiated in terms of code examples. We initiate the reduction by considering all repositories on GitHub. Then, the first query aims to identify repositories that use *EMF*. Many repositories exist that use *EMF* to varying extent. Here, we focus on "vanilla" *EMF* usage [HHL18]. Thus, repositories need to be excluded when they contain a mix of different technologies, such as *Xtext* or *acceleo*, which may distort the technology usage. For example, *EMF* code generation is influenced by the use of *Xtext*, since the code (and an Ecore model) is derived from a grammar. For inspecting more general or advanced usage of *EMF*, such exclusion criteria can be dropped. Table 7 summarizes this initial reduction that results in the top ten repositories selected based on their star rating.

In analogy to debugging code, we search for potentially missed code examples to optimize the accuracy of queries. For example, we search for code examples of the type `Interface` that are not the target of any `Generate Java Code` relation. The

ID	*Step*	*In*	*Out*	*Automation*
1	Formulate Query	*Expertise*: What co-technologies distort vanilla usage	*Query*: QegaL query described in [HHL18] based on file endings e.g. excluding projects with '.xtext'	M
2	Execute Query	*Corpus Resource*: GitHub *Query*: See **1**	*Query Results*: 1438 repos list available online	A
3	Link Evidence	*Query results:* See **2**	*Linked*: top ten sorted by stars see Table 8	M
4	Execute Query	*Corpus Resource*: Linked repos from step **3** *Query*: see Table 6 for an overview	*Query Results*: see Table 8	A
...	...	...	...	...

Table 7 – Excerpt of the reduction step protocol for the wild projects. It presents how the set of all repositories on GitHub is reduced to vanilla *EMF* repositories [HHL18].

respective query is provided in Listing 4. It is executed on the triplestore that results from executing the developed queries that return links to code examples.

By sampling based on the integrity of code examples, we find one generator model that refers to a '.uml' file instead of an Ecore model (see *R:EtoG*, column [3], in Table 8). As a result, the queries that are summarized in Figure 5 cannot match any Ecore package. Hence, the relation *GenerateJavaCode* (*R:GtoJ* in Table 8) cannot be instantiated (see *R:GtoJ*, column [3] in Table 8). Because we focus on idiomatic usage, where a generator model is based on an Ecore model instead of a '.uml' model, the query is not refined. Moreover, we also find repositories, where no model interfaces that extend *EObject* are recognized at all.

```
SELECT DISTINCT ?i WHERE {
   ?i sl:instanceOf sl:Interface .
   FILTER NOT EXISTS{ ?g sl:GenerateJavaCode ?i. }
}
```

Listing 4 – SPARQL queries for sampling query results to refine queries.

In Table 8, we present the results of querying selected GitHub projects to identify code examples of the technology model. We choose the top ten repositories sorted by star rating for further inquiry and sort them by their name. Several empty cells exist especially for the Java code-related columns, where we were unable to recover code examples for the Java artifact types. Still, the table shows that code examples of the technology model exist in multiple repositories.

4.1.5 Summary

We have presented and recorded how the methodology is executed for a technology model that depicts common *EMF* usage. Textual explanations are identified using *CTRL+F*; in demo projects, code examples are linked first to develop queries in a

Type /**R**elation	*Links* [1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	*Rationale*
T:EM	1	1	1	3	2	2	2	1	1	2	Ecore Model Query (see Listing 1)
T:GM	1	2	1	5	2	2	2	1	1	2	Generator Model Query (see Listing 1)
R:EtoG	1	1		5	2	1	2	1	1	2	Foreign Model Query (see Listing 2)
T:Int	111	2	2	12				46	6	5	Extend EObject Queries (see Figure 5)
T:Impl	82	2	2	15				46	6	4	Implements Queries (see Figure 5)
T:AF	3	1	1	4				1	1	2	Package Reference Query (see Figure 5)
R:GtoJ	167	5		38				93	13	4	Reference Query (see Figure 5)

Table 8 – Quantified links to ten vanilla *EMF* repositories (see column [1]-[10]). Abbreviations: EM=Ecore Model, GM=Generator Model, EtoG=Create Generator Model, Int=Interface, Impl=Interface Implementation, AF=Adapter Factory, GtoJ=Generate Java Code.

test driven manner; in wild projects, the queries are executed to find more complex code examples and check their commonness.

4.2 Revealing Misconception

In the second application of our methodology, we demonstrate how the execution of the methodology is robust enough to prevent misconception. We draw an example from our experience of teaching and conducting research on *EMF* usage. The concept of different model layers is often hard to understand for *newcomers*, which includes *students* and inexperienced *developers*. The technology model in Figure 6 summarizes a hypothetical misconception on the generated Model API. We summarize what is depicted as follows. Any generated model API consists of model classes written in Java. A student may explain the last relation as follows: 'Every model class is an EClass' while using the relation *subtype of* instead of *instance of*.

Before executing the reduction steps, we explain the misconception in this model. The technology model states that every model class inherits from EClass. Hence, a model class, which is at the level of metamodels, inherits from a class that is part of the metametamodel. Scientific literature clarifies the correct alignment of *EMF* terminology to the common model layers (model, metamodel, and metametamodel) [BBC+05, BHJ+05, GNF12].

4.2.1 Developer Literature

We again consider the book by Steinberg et al [SBMP08]. **Formulate Query:** We specifically focus on the extends relation with the queries "extends EClass", "subclass

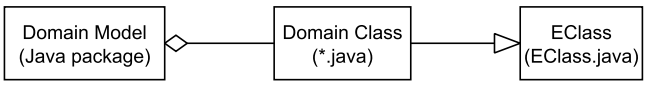

Figure 6 – This technology model summarizes a potential misconception on *EMF*.

of EClass", and "subtype of EClass", and just for "EClass". **Execute Query:** We execute the queries for the full text, because no result is returned when we apply the query to the table of contents. For the first three queries, no result can be found in the full text as well; for the last, we find out that Section 5 contains many query results. It explains the Ecore metametamodel that includes *EClass* as a class, but it does not suggest extending this class in a metamodel. **Link Evidence:** No textual explanation can be found in the query results. Either, the modeled technology usage is too advanced to be covered in the book. Or, it is just not intended by the *contributors*.

4.2.2 Scientific Literature

Next, we consolidate Google Scholar. **Formulate Query:** We use the same search strings that we used for developer literature, but we add 'EMF model' as a prefix and add surrounding quotes so that exact matches are retrieved, e.g., 'EMF model "subtype of EClass"'. **Execute Query:** Only one result is returned for "extends EClass"; a student's report project reports about problems when extending *EClass* [Sai10]. In analogy, only one result is returned for "subclass of EClass", a PhD thesis [Riv10] formalizes laws between instances of *EClass*. For the third query "subtype of EClass", four results are returned. **Link Evidence:** From the last query we find papers on *EMF* profiles [LWWC11, LWWC12]. They describe how *EMF* is adapted in order to introduce *stereotypes* as a concept at the metametamodel layer. Since, we avoid papers that describe adaption, no paper is linked.

4.2.3 Demo Projects

We take all demo projects as the corpus. **Formulate Query:** We search for classes that contain "extends EClass" in the class signature. **Execute Query:** No results are returned. We cannot draw any new conclusion from this, since the examples are aligned with the book's text passages.

4.2.4 Wild Projects

We use the GitHub search API for query execution. **Formulate Query:** We search for classes that contain "extends EClass" in the type signature. **Execute Query:** As expected, not many repositories exist. The code search returns 1,041 code examples. **Link Evidence:** The results almost always include `(<? extends EClass>)`. We exclude such cases. We find actual examples in an implementation of Petri Nets[5], where an interface called *Node* extends `EClass`. In this repository, not every interface extends EClass. Subtyping EClass is only used in very rare occasions. For the identified code examples, we assume that they are related to the identified scientific literature on *EMF* profiles [LWWC11, LWWC12]. The repository is not linked in the

[5] `github.com/cmodw/emf-profiles_testpetrinet`

papers, but there exists a respective repository with more code examples on *EMF* profiles for the same user[6].

4.2.5 Summary

In summary, we presented the execution of the methodology for a misconception in a technology model, for which no textual explanations can be found in developer literature. We identify textual explanations in scientific literature that describe a modification of EMF's metametamodel. The lack of code examples in demo projects and wild projects implies that the included extends relation does not represent idiomatic usage. In a hypothetical construction process, it is removed from the technology model.

5 Limitations

The evaluation in Section 4 is based on 'validation by examples' [Sha03] with the authors of this paper as executors. Thus, we essentially answered the research question of Section 3 by the provision of a methodology for constructing a technology model in a reproducible manner so that it is interconnected with textual explanations and code examples and then validating the methodology by examples.

This approach did not allow us to explore the limitations of the methodology, e.g., its dependence on experience of those executing the steps. Ultimately, controlled experiments – in which subjects execute the methodology for given tasks – could be expected to be useful in evaluating the methodology more thoroughly.

In this section, we sketch a *hypothetical* controlled experiment to make explicit a few hypotheses underlying the proposed methodology and to reveal its potential limitations. We would like to emphasize that the completion of this sketch into an actual experiment design and performing the experiment would represent a very significant challenge, but we contend that the discussion of the hypothetical setup is insightful nevertheless.

In a controlled experiment, one would present tasks to the participants so that they need to connect a given technology model with textual explanations and code examples as presented in Section 4. As the dependent variable, one measures *time* needed by participants for task completion. In the analysis, one is interested in what variables influence the time.

Experience The first independent variable is the *experience* of a participant. This paper's validation is limited in so far that we did not involve stakeholders with different levels of experience who execute our methodology. Experience is difficult to objectively measure. It is typically recorded in a questionnaire that captures a subjective estimate before the experimented is started. We hypothesize that the time depends on the given *experience*. If precise textual explanations and idiomatic code examples can be linked without any querying effort, the time is at a minimum. If no query can be formulated from the beginning, the time is at a maximum.

Complexity The second independent variable is the *complexity of the technology model*. This paper's validation is limited in so far that we did not execute the methodology for constructing a broad range of technology models. (The different facets of the methodology were based though on separate megamodeling efforts, as cited in the

[6] `github.com/cmodw`

paper.) To cover more complex aspects, it is harder to find textual explanations and code examples. We hypothesize that more time is needed to find links, because more queries and more complex queries have to be developed.

Quality of Resources The third independent variable is the *quality* of textual explanations and code examples. This paper's validation is limited in so far that we made subjective decisions on what is added as an increment and what evidence is linked. An external gold standard is still abundant and can only be gained by efforts of an *experienced* community, e.g., by involving *EMF* contributors. A related issue is that the set of linked resources for a given technology model is not uniquely defined, unless we highly constrain the underlying corpora. Thus, the task results produced by the subjects in a controlled experiment would need to be checked for correctness and possibly graded, which does not appear to be straightforward.

6 Related Work

Megamodeling Technology Usage We draw our main inspiration from related work in linguistic architecture of software starting with its introduction in [FLV12]. There, we suggest using language-centric megamodels to describe software applications and capture the use of different technologies and languages. Its semantics and interpretation is then discussed in [LV14a]. In [HLV17], we describe a survey of megamodelling literature that summarizes common vocabulary; we also axiomatize some types and relationships used in megamodels for technology usage. In [HHL+17], more tool support and conceptual principles for linguistic architecture models are introduced, where links between megamodel elements (entities and relations) and code are systematically supported. In [Zay12, LZ13, Zay14], megamodels are 'renarrated', i.e., they are presented in a sequential, story-telling manner. In this previous work, scattered elements of analysis of demo projects, wild projects, developer literature, and scientific literature show up. In the present paper, we generalize these experiences into a methodology for interconnecting resources within technology models.

Technology Usage Related work interested in technology usage is diverse. Under the premise of better understanding state-of-the-art practices, related work covers the systematic analysis of MDE technology usage in open source repositories [HHL18, RRH+18, KMK+15]. Similar analysis has been conducted for large industrial settings [MSSvdB17]. The different shapes of the Model-View-Controller model has been analyzed in different technologies in [DD19]. Research focused on API usage examines, for instance, sequences of method calls [RBK+13], type usage [ZM19], multi-level patterns [SBAS15] or code examples [LDZ19]. The textual explanations of APIs are subject to research conducted in [PRM15]. The usage of graph query languages is examined in [SHL+19]. The usage of databases in Android application has been discussed in [LGWH17]. The evolution of testing library usage has been subject to research conducted in [ZM17]. However, such related work does not come up with a methodology like ours, because typically such work does not aim at modeling technology usage. Opposed to the previously listed work, our paper focuses on technology usage in terms of artifacts and their relations. We aim at constructing a technology model supported by the usage analysis of a particular technology.

Ontology Evaluation We are also inspired by research on ontology evaluation. Gold standard-based evaluation is a prominent method in this field [DS06]. In our case, we cannot make use of this method, because a gold standard does not exist yet. In a corpus-based ontology evaluation [RTSP12], an ontology is evaluated based on how well it covers a given corpus. We can draw analogies to this method, because we are interested in covering corpora of code examples as well as textual explanations.

7 Conclusion

Summary We have provided a methodology for the reproducible construction of technology models that are interconnected with textual explanations and code examples. Such interconnected models can assist *newcomers* and *developers* at understanding technology usage and help *teachers* and *contributors* to share and extend their *experience*. We envisage that the reproducible construction of interconnected technology models can be used to provide a systematic approach to communicating textual explanations and code examples. It complements existing forms of patterns that assist at understanding complex software [PGG+15].

We start from an initial (possibly incomplete and incorrect) technology model, without interconnections, which is set up as a 'working assumption'. The technology model is subsequently linked to valuable textual explanations and code examples so that it becomes more than just a visual summary. To this end, developer literature, scientific literature, demo projects, and wild projects are processed. Artifact types and relations are confirmed or rejected based on manually or automatically executed queries. Evidence is then integrated into the model as links. Importantly, the methodology features a reduction procedure that is aimed at minimizing the manual effort for identifying and then linking concise textual explanations and idiomatic code examples.

Future Work This paper covers initial efforts on modeling the *EMF* technology. There are clearly more *EMF* usage aspects that one may want to discuss in detail, but we selected code generation to cover important basics of *EMF*. The longer-term research goal (with regard to *EMF*) is to construct a reference model of EMF usage as a gold standard of interconnected text and code resources. A complete technology model for *EMF* is to be expected only by an iterative process that involves the community to further decrease subjectivity introduced by manual steps. Technology models need to be continuously challenged and revised.

Beyond the scope of *EMF*, we would like to see that the proposed methodology can be shown to help with broader efforts on developing a comprehensive ontology in software engineering [Gon15], and more specifically, the software language engineering body of knowledge (SLEBOK) [CLW17]. To this end, further refinements of the proposed methodology may be needed as well as additional forms of validation, as discussed in Section 5. We hypothesize that exploring interconnected technology models with dedicated tool support, for example, similar to [RLP13], may help in understanding *EMF*, and possibly other complex technology, such as *Xtext*.

References

[BB13] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *Proc. ICSE*, pages 712–721,

2013.

[BBC+05] Jean Bézivin, Christian Brunette, Régis Chevrel, Frédéric Jouault, and Ivan Kurtev. Bridging the generic modeling environment (gme) and the eclipse modeling framework (emf). In *Proc. OOPSLA*, 2005.

[BEJ10] Enrico Biermann, Claudia Ermel, and Stefan Jurack. Modeling the "ecore to genmodel" transformation with emf henshin. *Transformation Tool Contest*, page 153, 2010.

[BHJ+05] Jean Bézivin, Guillaume Hillairet, Frédéric Jouault, Ivan Kurtev, and William Piers. Bridging the ms/dsl tools and the eclipse modeling framework. In *Proc. OOPSLA*, 2005.

[BS15] Thomas Buchmann and Felix Schwägerl. On a-posteriori integration of ecore models and hand-written java code. In Pascal Lorenz, Marten van Sinderen, and Jorge S. Cardoso, editors, *Proc. ICSOFT-PT*, 2015.

[BY06] Mordechai Ben-Ari and Tzippora Yeshno. Conceptual models of software artifacts. *Interacting with Computers*, 18(6):1336–1350, 2006.

[CLW17] Benoît Combemale, Ralf Lämmel, and Eric Van Wyk. SLEBOK: the software language engineering body of knowledge (dagstuhl seminar 17342). *Dagstuhl Reports*, 7(8):45–54, 2017.

[DD19] Dragos Dobrean and Laura Diosan. An analysis system for mobile applications MVC software architectures. In *Proc. ICSOFT*, 2019.

[DS06] Klaas Dellschaft and Steffen Staab. On how to perform a gold standard based evaluation of ontology learning. In *PRoc. ISWC*, pages 228–241, 2006.

[EIG+15] Hamza Ed-Douibi, Javier Luis Cánovas Izquierdo, Abel Gómez, Massimo Tisi, and Jordi Cabot. EMF-REST: Generation of RESTful APIs from Models. *CoRR*, abs/1504.03498, 2015.

[FLV12] Jean-Marie Favre, Ralf Lämmel, and Andrei Varanovich. Modeling the linguistic architecture of software products. In *Proc. MODELS*, pages 151–167, 2012.

[GGKdL14] Antonio Garmendia, Esther Guerra, Dimitrios S. Kolovos, and Juan de Lara. EMF Splitter: A Structured Approach to EMF Modularity. In *XM@MoDELS*, 2014.

[GNF12] José Manuel Gascueña, Elena Navarro, and Antonio Fernández-Caballero. Model-driven engineering techniques for the development of multi-agent systems. *Eng. Appl. of AI*, 25(1):159–173, 2012.

[Gon15] Cesar Gonzalez-Perez. How ontologies can help in software engineering. In *Proc. GTTSE*, 2015.

[Hea92] Marti A. Hearst. Automatic acquisition of hyponyms from large text corpora. In *Proc. COLING*, pages 539–545, 1992.

[HGG12] Regina Hebig, Gregor Gabrysiak, and Holger Giese. Towards patterns for mde-related processes to detect and handle changeability risks. In *Proc. ICSSP*, pages 38–47, 2012.

[HHL+17] Johannes Härtel, Lukas Härtel, Ralf Lämmel, Andrei Varanovich, and Marcel Heinz. Interconnected linguistic architecture. *Programming Journal*, 1(1):3, 2017.

[HHL18] Johannes Härtel, Marcel Heinz, and Ralf Lämmel. EMF patterns of usage on github. In *Proc. ECMFA*, pages 216–234, 2018.

[HLV17] Marcel Heinz, Ralf Lämmel, and Andrei Varanovich. Axioms of linguistic architecture. In *Proc. MODELSWARD*, pages 478–486. SCITEPRESS, 2017.

[KBB+09] Barbara A. Kitchenham, Pearl Brereton, David Budgen, Mark Turner, John Bailey, and Stephen G. Linkman. Systematic literature reviews in software engineering - A systematic literature review. *Information & Software Technology*, 51(1):7–15, 2009.

[KGRP17] Dimitrios S. Kolovos, Antonio García-Domínguez, Louis M. Rose, and Richard F. Paige. Eugenia: towards disciplined and automated development of gmf-based graphical model editors. *Software and Systems Modeling*, 16(1):229–255, 2017.

[KMK+15] Dimitrios S. Kolovos, Nicholas Drivalos Matragkas, Ioannis Korkontzelos, Sophia Ananiadou, and Richard F. Paige. Assessing the use of eclipse MDE technologies in open-source software projects. In *OSS4MDE@MoDELS*, 2015.

[KRbA+10] Dimitrios S. Kolovos, Louis M. Rose, Saad bin Abid, Richard F. Paige, Fiona A. C. Polack, and Goetz Botterweck. Taming EMF and GMF using model transformation. In Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *Proc. MODELS*, 2010.

[LDZ19] Binbin Liu, Wei Dong, and Yinzhu Zhang. Accelerating api-based program synthesis via API usage pattern mining. *IEEE Access*, 7:159162–159176, 2019. doi:10.1109/ACCESS.2019.2950232.

[LGWH17] Yingjun Lyu, Jiaping Gui, Mian Wan, and William G. J. Halfond. An Empirical Study of Local Database Usage in Android Applications. In *ICSME*, pages 444–455. IEEE Computer Society, 2017.

[LSV13] Ralf Lämmel, Thomas Schmorleiz, and Andrei Varanovich. The 101haskell chrestomathy: A whole bunch of learnable lambdas. In *Proc. IFL*, page 25, 2013.

[LV14a] Ralf Lämmel and Andrei Varanovich. Interpretation of linguistic architecture. In *Proc. ECMFA*, pages 67–82, 2014.

[LV14b] Ralf Lämmel and Andrei Varanovich. Interpretation of Linguistic Architecture. In *Proc. ECMFA 2014*, volume 8569 of *LNCS*, pages 67–82. Springer, 2014.

[LWWC11] Philip Langer, Konrad Wieland, Manuel Wimmer, and Jordi Cabot. From UML profiles to EMF profiles and beyond. In *Proc. TOOLS*, pages 52–67, 2011.

[LWWC12] Philip Langer, Konrad Wieland, Manuel Wimmer, and Jordi Cabot. EMF profiles: A lightweight extension approach for EMF models. *Journal of Object Technology*, 11(1):1–29, 2012.

[LZ13] Ralf Lämmel and Vadim Zaytsev. Language support for megamodel renarration. In *Proc. Extreme Modeling co-located with MODELS (2013)*, pages 36–45, 2013.

[MBSJ09] Mike Mintz, Steven Bills, Rion Snow, and Daniel Jurafsky. Distant supervision for relation extraction without labeled data. In Keh-Yih Su, Jian Su, and Janyce Wiebe, editors, *Proc. ACL*, pages 1003–1011, 2009.

[MSSvdB17] Josh GM Mengerink, Alexander Serebrenik, Ramon RH Schiffelers, and Mark GJ van den Brand. Automated analyses of model-driven artifacts: obtaining insights into industrial application of mde. In *Proc. IWSM-Mensura*, pages 116–121. ACM, 2017.

[PGG+15] Ana Pescador, Antonio Garmendia, Esther Guerra, Jesús Sánchez Cuadrado, and Juan de Lara. Pattern-based development of domain-specific modelling languages. In *Proc. MoDELS*, pages 166–175, 2015.

[PRM15] Gayane Petrosyan, Martin P. Robillard, and Renato De Mori. Discovering information explaining API types using text classification. In *Proc. ICSE*, pages 869–879, 2015.

[RBK+13] Martin P. Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. Automated API property inference techniques. *IEEE Trans. Software Eng.*, 39(5):613–637, 2013.

[RC15] Martin P. Robillard and Yam B. Chhetri. Recommending reference API documentation. *Empirical Software Engineering*, 20(6):1558–1586, 2015.

[Riv10] José Eduardo Rivera. *On the semantics of real-time domain specific modeling languages*. PhD thesis, Ph. D. thesis, Universidad de Málaga, 2010.

[RLP13] Coen De Roover, Ralf Lämmel, and Ekaterina Pek. Multi-dimensional exploration of API usage. In *Proc. ICPC*, pages 152–161. IEEE Computer Society, 2013.

[Rob09] Martin P. Robillard. What makes apis hard to learn? answers from developers. *IEEE Software*, 26(6):27–34, 2009.

[RRH+18] Juri Di Rocco, Davide Di Ruscio, Johannes Härtel, Ludovico Iovino, Ralf Lämmel, and Alfonso Pierantonio. Systematic recovery of MDE technology usage. In *Proc. ICMT*, pages 110–126, 2018.

[RTSP12] Marco Rospocher, Sara Tonelli, Luciano Serafini, and Emanuele Pianta. Corpus-based terminological evaluation of ontologies. *Applied Ontology*, 2012.

[Sai10] David Saile. Integrating twouse and ocl-dl, 2010. Student project. University of Koblenz-Landau.

[SBAS15] Mohamed Aymen Saied, Omar Benomar, Hani Abdeen, and Houari A. Sahraoui. Mining Multi-level API Usage Patterns. In *SANER*, pages 23–32. IEEE Computer Society, 2015.

[SBMP08] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[Sha03] Mary Shaw. Writing good software engineering research paper. In Lori A. Clarke, Laurie Dillon, and Walter F. Tichy, editors, *Proc. ICSE*, pages 726–737. IEEE Computer Society, 2003.

[SHL+19] Philipp Seifer, Johannes Härtel, Martin Leinberger, Ralf Lämmel, and Steffen Staab. Empirical study on the usage of graph query languages in open source Java projects. In *SLE*, pages 152–166. ACM, 2019.

[SLH+17] Simon Schauss, Ralf Lämmel, Johannes Härtel, Marcel Heinz, Kevin Klein, Lukas Härtel, and Thorsten Berger. A chrestomathy of DSL implementations. In *Proc. SLE*, pages 103–114, 2017.

[Zay12] Vadim Zaytsev. Renarrating linguistic architecture: a case study. In *Proc. MPM@MoDELS 2012*, pages 61–66. ACM, 2012.

[Zay14] Vadim Zaytsev. Understanding Metalanguage Integration by Renarrating a Technical Space Megamodel. In *Proc. GEMOC@Models*, 2014.

[ZM17] Ahmed Zerouali and Tom Mens. Analyzing the evolution of testing library usage in open source Java projects. In *SANER*, pages 417–421. IEEE Computer Society, 2017.

[ZM19] Hao Zhong and Hong Mei. An Empirical Study on API Usages. *IEEE Trans. Software Eng.*, 45(4):319–334, 2019.

About the authors

Marcel Heinz Marcel Heinz is a research assistant in the software languages team at the University of Koblenz-Landau. He is working towards a Ph.D. His general interests are in Software Language Engineering, Modeling and Knowledge Engineering. His research focus is on assisting at the comprehension of software languages and software technologies. Contact him at heinz@uni-koblenz.de or visit `http://www.softlang.org`.

Johannes Härtel Johannes Härtel received his MSc in Computer Science from the University of Koblenz-Landau Germany in 2016 where he is currently working as researcher. He is doing his PhD on tool and language support for mining software repositories. His research interests include mining software repositories, software language engineering, natural language processing, functional programming and model-based techniques. Contact him at johanneshaertel@uni-koblenz.de or visit `http://www.softlang.org`.

Ralf Lämmel Ralf Lämmel is Software Engineer at Facebook since 2018 and Professor of Computer Science (currently on leave of absence) at the University of Koblenz-Landau in Germany since 2007. In the past, he had held positions at the University of l'Aquila, Microsoft, the Free University of Amsterdam, CWI (Dutch Center for Mathematics and Computer Science), and the University of Rostock, Germany.
His research and teaching interests include software language engineering, software reverse engineering, software re-engineering, mining software repositories, functional programming, grammar-based and model-based techniques, and, more recently, megamodeling. In his current work at Facebook, he applies machine learning (in a broad sense) in an infrastructural context while developing an increasing interest in data engineering and science.
He is one of the founding fathers of the international summer school series on Generative and Transformational Techniques on Software Engineering (GTTSE) and the international conference on Software Language Engineering (SLE). He is the author of Springer textbook on Software Language Engineering: Software Languages: Syntax, Semantics, and Metaprogramming, Springer, 2018, which received the Choice Award "Outstanding Academic Title" in 2019. Contact him at laemmel@uni-koblenz.de or visit `http://www.softlang.org`.