Knowledge Engineering for Software Languages and Software Technologies

Marcel Heinz

Universität Koblenz-Landau
Fachbereich 4: Informatik

Genehmigte Dissertation
zur Verleihung des akademischen Grades
eines Doktors der Naturwissenschaften (Dr. rer. nat.),
Fachbereich 4: Informatik, Universität Koblenz-Landau

Vorsitzende des Promotionsausschusses:	Prof. Dr. Karin Harbusch
Berichterstatter:	Prof. Dr. Ralf Lämmel
	Associate Prof. Davide di Ruscio
	Associate Prof. Mathieu Archer

Datum der wissenschaftlichen Aussprache: 14.09.2021

Abstract

For software engineers, conceptually understanding the tools they are using in the context of their projects is a daily challenge and a prerequisite for complex tasks. Textual explanations and code examples serve as knowledge resources for understanding software languages and software technologies. This thesis describes research on integrating and interconnecting existing knowledge resources, which can then be used to assist with understanding and comparing software languages and software technologies on a conceptual level. We consider the following broad research questions that we later refine: *What knowledge resources can be systematically reused for recovering structured knowledge and how? What vocabulary already exists in literature that is used to express conceptual knowledge? How can we reuse the online encyclopedia Wikipedia? How can we detect and report on instances of technology usage? How can we assure reproducibility as the central quality factor of any construction process for knowledge artifacts?* As qualitative research, we describe methodologies to recover knowledge resources by i.) systematically studying literature, ii.) mining Wikipedia, iii.) mining available textual explanations and code examples of technology usage. The theoretical findings are backed by case studies. As research contributions, we have recovered i.) a reference semantics of vocabulary for describing software technology usage with an emphasis on software languages, ii.) an annotated corpus of Wikipedia articles on software languages, iii.) insights into technology usage on GitHub with regard to a catalog of pattern and iv.) megamodels of technology usage that are interconnected with existing textual explanations and code examples.

Zusammenfassung

Softwaresprachen und Technologien zu verstehen, die bei der Entwicklung einer Software verwendet werden, ist eine alltägliche Herausforderung für Software Engineers. Textbasierte Dokumentationen und Codebeispiele sind typische Hilfsmittel, die zu einem besseren Verständnis führen sollen. In dieser Dissertation werden verschiedene Forschungsansätze beschrieben, wie existierende Textpassagen und Codebeispiele identifiziert und miteinander verbunden werden können. Die Entdeckung solcher bereits existierender Resourcen soll dabei helfen Softwaresprachen und Technologien auf einem konzeptionellen Level zu verstehen und zu vergleichen.

Die Forschungsbeiträge fokussieren sich auf die folgenden Fragen, die später präzisiert werden. *Welche existerenden Ressourcen lassen sich systematisch identifizieren, um strukturiertes Wissen zu extrahieren? Wie lassen sich die Ressourcen extrahieren? Welches Vokabular wird bereits in der Literatur verwendet, um konzeptionelles Wissen zur Struktur und Verwendung einer Software auszudrücken? Wie lassen sich Beiträge auf Wikipedia wiederverwenden? Wie können Codebeispiele zur Verwendung von ausgewählten Technologien auf GitHub gefunden werden? Wie kann ein Modell, welches Technologieverwendung repräsentiert, reproduzierbar konstruiert werden?*

Zur Beantwortung der Forschungsfragen werden qualitative Forschungsmethoden verwendet wie zum Beispiel Literaturstudien. Des weiteren werden Methoden entwickelt und evaluiert, um relevante Artikel auf Wikipedia, relevante Textpassagen in der Literatur und Codebeispiele auf GitHub zu verlinken. Die theoretischen Beiträge werden in Fallstudien evaluiert.

Die folgenden wissenschaftlichen Beiträge werden dabei erzielt: i.) Eine Referenzsemantik zur Formalisierung von Typen und Relationen in einer sprachfokussierten Beschreibung von Software; ii.) Ein Korpus bestehend aus Wikipedia Artikeln zu einzelnen Softwaresprachen; iii) Ein Katalog mit textuell beschriebenen Verwendungsmustern einer Technologie zusammen mit Messergebnissen zu deren Frequenz auf GitHub; iv.) Technologiemodelle, welche sowohl mit verschiedenen existierenden Codebeispielen als auch mit Textpassagen verknüpft sind.

Short Biography

April 2010 - June 2014 B.Sc. Computer Science at University of Koblenz-Landau.

June 2014 - March 2015 M.Sc. Computer Science at University of Koblenz-Landau.

April 2015 - March 2020 Research Assistant at University of Koblenz Landau.

June 2020 - ? Developer at Senacor Technologies AG.

Marcel Heinz was a research assistant from 2015 to 2020 in the software languages team at the University of Koblenz-Landau. He defended his thesis on 14th of September 2021. His general interests are in Software Language Engineering, Modeling, Knowledge Engineering and Data Engineering. His research focus is on assisting at the comprehension of software languages and software technologies.

Acknowledgements

This thesis is based on more than five years of continuous research. The goals and the methodologies have evolved and changed directions several times over the years. With the following sentences, I would like to express my gratitude to my companions on my way and reflect on the path that we have walked on together. First of all, I am grateful to my parents for pushing me when it was needed and for enabling me to identify and dive into a passion, which is software engineering. The kind of support that only seems like little deeds at first sight has been so valuable for so many times. Most of all, I would like to thank my girlfriend at researcher times for calming me down when I was stressed out, for sharing the passion for software engineering, for every little piece of chocolate that you gave me when times were rough and for always being there for me. I am thankful to my mentor Ralf Lämmel for shaping diverse skill sets. Together with Johannes, the passion for software engineering resounded. This lead to several publications based on hard work as well as pizza and wine. While working as a software engineer in industry, I have been reminded that you did not only teach me how to conduct research. Other skill sets evolved maybe even without your explicit intention. I am happy to be blessed with companions in my private life, who helped me to stay sane. You know who you are: the “Wohngemeinschaft”, the coffee, the one and only essential night club, the sushi, the poker games, the billiard games, the tsukis, the tooth brushing, the golden sands, the wall’s destruction, the couch, and more.

Contents

Chapter 1

Introduction

1.1 Research Context

In any software project, software engineers need to conceptually understand the used software languages and software technologies. Gaining and sharing knowledge is a prerequisite to the successful completion of complex software projects. Therefore, software engineers consolidate heterogeneous knowledge resources on a daily basis.

In this thesis, we describe how to reuse existing knowledge resources, e.g., textual explanations and code examples, that help with understanding software languages and software technologies on a conceptual level and with respect to how they are used throughout software projects. We conduct multi-disciplinary research that relates to known approaches on knowledge discovery, data classification, ontologies, software engineering, reverse engineering, and model-driven engineering. In the following subsections, we outline the topics that are central to our research contributions. This thesis can be seen as a continuation of existing research, e.g., [Favre et al., 2012c, Lämmel et al., 2013a, Härtel et al., 2017, Varanovich, 2018]. Therefore, we also refer to previous research that largely emphasizes a software language-centered view on software.

1.1.1 Software Technology

The huge amount of existing software technologies creates a demand for organizing knowledge resources. Software technologies can be categorized by technological spaces [Kurtev et al., 2002], e.g., XML-ware or SQL-ware. Each technological space has its own descriptions of specific problems and technical solutions. As motivated in [Smith and Welty, 2001], the complexity of software engineering and closely related fields calls for the application of ontology engineering. Related approaches exist that focus mainly on API (method) usage [Robillard et al., 2013].

The Softlang team has been maintaining the software chrestomathy '101companies' [Favre et al., 2012b, Lämmel, 2015, Varanovich, 2018][1] which is a collection of small software systems that implement a common feature model while aiming to represent the best practices and variety of language usage, technology usage, and software design. The systems are documented on a semantic wiki called *101wiki* at contribution pages; the documentation includes semantic triples the used software languages and software technologies. In *101wiki*, chrestomathies for Haskell [Lämmel et al., 2013c] and domain-specific language engineering [Schauss et al., 2017] are maintained as well.

[1] `http://101companies.org` — Requested March 31, 2022

1.1.2 Software Language

According to literature, a software language is any artificial language that is used for developing software [Lämmel, 2018, Favre et al., 2009, Kleppe, 2008]. We consider it to be synonymous for computer language. A dedicated discussion on the synonymy is provided in [Favre et al., 2009]. A software language is not only used to communicate with the computer. Software engineering today is also about communicating with other fellow software engineers. With a focus on programming languages, the notion of a paradigm as a pattern-like way of thinking is introduced in Floyd's Turing award lecture [Floyd, 1979]. Each paradigm supports a set of concepts that are necessary for the solution to a problem. A guide to choosing paradigms and a programming language that supports them based on a problem description is offered in [van Roy, 2009]. Classifying and comparing software languages is more complex and cannot be solved by considering paradigms alone [Shilov et al., 2012]. Flexibility and naturalness of syntax should also be considered. In [Shilov et al., 2012], the creation of a collaboratively maintained flexible computer language ontology is described.

Software languages are central to the notion of a linguistic architecture [Favre et al., 2012c]. Here, they are used as linguistic properties of artifacts in a model of a software project and technology usage scenarios. In related work on software chrestomathies [Favre et al., 2012b, Lämmel, 2015, Varanovich, 2018], software languages are annotated as concepts to tell which languages are used by a contribution. Wikipedia contains many articles on software languages that mainly exist within the category 'Computer languages'. In [Lämmel et al., 2013b], Wikipedia categories relevant for the classification of software languages are identified through a proposed interactive tool.

1.1.3 Wikipedia

Wikipedia has a rich information ecosystem with many difficulties that have to be managed first. Related research aims to recover knowledge from the various types of contents in Wikipedia articles, such as the title [Zarrad et al., 2013], first sentence [Flati et al., 2014], text [Rios-Alvarado et al., 2011, Presutti et al., 2014], section structure [Wang et al., 2010], links to other articles [Nuzzolese et al., 2011, Presutti et al., 2014], infoboxes [Wu and Weld, 2008], and lists [Kuhn et al., 2016].

The Softlang Team has been reusing knowledge from Wikipedia in multiple occasions. In *101wiki* that is introduced in [Favre et al., 2012b], pages documenting conceptual entities such as a software language often refer to articles on Wikipedia. In a recently published chrestomathy on DSL implementations Metalib [Schauss et al., 2017], the contributions are classified by concepts, which are then linked to the *101wiki*. New concepts were added to *101wiki* that are described on Wikipedia or sometimes even in linked papers. In [Lämmel et al., 2013b], a tool is proposed to prune Wikipedia's category graph by manually excluding categories not serving for classification of software languages in a common sense, subject to a small suite of criteria. For instance, a category may be observed as serving maintenance (e.g., "Uncategorized programming languages") or as encoding properties (e.g., "Discontinued programming languages") and is hence abandoned.

1.1.4 Mining Software Repositories

Mining software repositories on is a common activity to understand how a technology is used in existing software [Kolovos et al., 2015, Rocco et al., 2018, Seifer et al., 2019]. Related research focuses on analyzing pattern of API usage [Saied et al., 2015], classifying API domains [Härtel et al., 2018a], or analyzing energy efficiency [Lyu et al., 2017]. In this thesis, we are mainly interested in identifying pattern of technology usage and their detection, which follows similar steps as the detection of design patterns [Roover, 2011].

In previous work by the Softlang Team, a tool is proposed to detect and explore API usage [Roover et al., 2013]. The classification of APIs by hierarchical clustering includes hints towards how APIs can be grouped and compared to each other [Härtel et al., 2018a]. More recently, in [Seifer et al., 2019], an empirical study is conducted to analyze the popularity of graph query languages.

1.1.5 Megamodeling

Megamodels treat models as first class entities and capture their relationships [Bézivin et al., 2004]. In this thesis, we investigate on different types of models and their relations. We, for example, inspect schema files and code object models as well as model types from the domain of model-driven engineering (MDE), such as UML[2] models.

In [Favre et al., 2012c], the term *linguistic architecture* is introduced as a description of software systems that emphasizes the classification of artifacts and processes by software languages. The Softlang team has been designing megamodeling languages for linguistic architecture, most notably MegaL[3] [Favre et al., 2012c]. The megamodels declare how 'digital' entities (such as files or objects) and 'conceptual' entities (such as languages or programming techniques) relate in the context of scenarios of technology and language usage. Such declarations can be verified [Lämmel and Varanovich, 2014] and used to analyze consistency aspects in software projects [Härtel et al., 2017].

1.2 Research Questions

Our overarching goal is to help software engineers with understanding the myriad of software languages and software technologies. The following subsections describe the research motivation and questions for our individual contributions as steps towards that goal. For each contribution, we motivate a problem to be solved and raise research questions that are answered in the chapters on individual contributions.

1.2.1 Axioms of Linguistic Architecture

Informally, conceptual relationships between artifacts are often described in technical documentation. They are only partly encoded in standard viewpoints covered by the UML. When encoding knowledge on the use of software languages in the context of software technology, one primary task is to provide a formal ground to the used vocabulary for knowledge representation. Central types, relations and formal axioms may together form a schematic body of knowledge that is often referred to as core ontology [Scherp et al., 2011]. Hence, we aim to recover such a schematic body of knowledge based on already existing documentation idioms.

Conceptual relations between artifacts are encoded in megamodels. We hypothesize that frequent names of type and relations can be identified in megamodeling literature. Thus, we face the challenge of discovering and aligning common vocabulary. To form a schematic body of knowledge, such vocabulary needs to be formally aligned in terms of formal axioms. We answer the research questions below in Chapter 3.

RQ A1: What types of entities and relations are common in megamodels?

RQ A2: What modeling idioms exist for (language-centric) megamodels that can be formalized as axioms?

[2] `https://www.omg.org/spec/UML/About-UML/` — Requested March 31, 2022
[3] `http://www.softlang.org/megal` — Requested March 31, 2022

1.2.2 Software Languages at Wikipedia

To recover and organize knowledge on software languages, which is central for research on linguistic architecture [Favre et al., 2012c, Favre et al., 2012b], Wikipedia can be used as an initial source of information. Many articles on Wikipedia describe properties of individual software languages. Only a few articles encode classification of the software language in infoboxes that serve as a summary. In others, a classification can be extracted from the article's text. Wikipedia has many peculiarities, which are described in its guidelines[4] and challenge automated extraction. Additionally, such guidelines are not automatically enforced. As a result, domains in Wikipedia may be managed to varying degrees. Such peculiarities have to be rigorously explored before an accurate extraction is possible. Hence, we state the first research question as follows:

> *RQ W0: What types of content in a single Wikipedia article can be used for extracting knowledge and what challenges need to be addressed?*

Discovering articles that describe instances of a specific ontological class such as the class *software language* is challenging. The guidelines allow and encourage authors to write one articles on multiple topics, when this benefits understandability. Initial explorations presented in Section 4.1 show that even in the category tree below 'Computer languages' relevant articles are rare. Hence, we answer the research question below that is refined in Chapter 4.

> *RQ W1: How can we classify Wikipedia articles by their relevance to a given domain when relevant articles are rare and multiple main topics are covered by articles?*

1.2.3 Mining Technology Usage

Aside from technical documentation, existing code examples can be inspected to understand how to use a software technology. Existing research typically focuses on the analysis of how a technology's API is used on the level of program elements [Roover et al., 2013, Robillard et al., 2013, Khatchadourian et al., 2020]. In this thesis, we are not only focusing on program elements, but are interested in other types of artifacts such as configuration models as well.

Many repositories implementing complex business logic with a specific technology, such as EMF, can be found on GitHub. The repositories range from student projects to open source projects, which are still active and maintained. Code in open source projects can be analyzed with regard to whether it instantiates patterns of technology usage. We answer the research question below in Chapter 5.

> *RQ T1: How can we locate traces of technology usage on GitHub?*

1.2.4 Modeling Technology Usage

Artifacts and their relationships have been modeled with respect to technology usage in the context of linguistic architecture [Favre et al., 2012c, Favre et al., 2012b, Härtel et al., 2017]. So far, linguistic architecture and as megamodels in general [Rocco et al., 2017], are typically focused on maintaining and describing single projects.

In this thesis, we are interested in the influence of using a specific technology on the design of *any* software project that uses it. We refer to such megamodels of technology usage as technology models. Based on our experience with modeling technology usage in prior efforts, we identify multiple challenges that need to be addressed: i.) We aim to assure that we model technology usage in a way that represents textual explanations in literature. ii.) We aim to

[4] `https://en.wikipedia.org/wiki/Wikipedia:Policies_and_guidelines` — Requested March 31, 2022

assure that what we model is actually common in software projects using the technology. Thus, we systematically search for code examples instantiating technology models. iii.) Other experts need to be able to reproduce how textual explanations and code examples have been identified by executing the same queries. Potentially, the same queries can be executed to identify textual explanations and code examples in resources that have not been used yet. The research question below is answered in Chapter 6.

> *RQ T2: How can we construct a technology model in a reproducible manner so that it is interconnected with existing textual explanations and code examples?*

1.3 Research Method

This section provides an overview on the research methodologies that are used for answering the research questions.

1.3.1 Axioms of Linguistic Architecture

Our goal is to develop a reference semantics for vocabulary that is used to describe software projects with regard to the used software languages and software technologies. Thereby, we develop a core ontology on Software Languages and Software Technologies (SoLaSoTe). By providing a formal basis, we hope to reach a high degree of understandability and consistency for knowledge encoded by a language-centric megamodel. This formal basis forms the vocabulary used to describe or prescribe usage of software languages and software technologies instead of relying on an intuitive understanding.

We hypothesize that the essential vocabulary already exists and is defined in literature. Therefore, we perform a *systematic literature analysis* [Petersen et al., 2008, Elberzhager et al., 2012]. We discover and align vocabulary from various published contributions in megamodeling literature. As a first step, we recover a corpus of research papers from 'ACM Digital Library' (ACM)[5], 'Springer Link' (Springer)[6] and 'IEEE Xplore Digital Library' (IEEE)[7] by using the sites' search engines with the search string *"'mega model" OR "mega-model" OR "megamodel"'*. Then, we exclude papers so that only those remain that describe vocabulary relevant to megamodels. From the filtered corpus, we inspect the text and conceptual diagrams and recover reoccurring types of entities and relations. As a second step, we carefully develop formal axioms that express documentation idioms. Such idioms can be found in literature and documentation of a technology. They provide grounds for validating the integrity [Tran and Debruyne, 2012] of megamodels. We exemplify recovered types and relations for Java code and models specific to EMF usage.

Thesis contribution: The contribution is published in [Heinz et al., 2017]. In Chapter 3, this thesis augments the published contribution as follows: i.) The systematic literature study is updated by considering recently published papers. ii.) The axioms are slightly modified to improve the structure of logic formulas. Changes to axioms are highlighted and described by dedicated footnotes. iii.) We add competency questions that motivate the formal axioms and help with understanding them and their intended meaning. iv.) The axioms are implemented in Prolog to demonstrate consistency and integrity. Hence, the published examples in [Heinz et al., 2017] are translated to Prolog knowledge bases, which are then validated against axioms.

---

[5] `http://dl.acm.org/` — Requested March 31, 2022
[6] `http://link.springer.com/` — Requested March 31, 2022
[7] `http://ieeexplore.ieee.org/Xplore/home.jsp` — Requested March 31, 2022

1.3.2 Wikipedia

Software languages and software technologies are described in articles of Wikipedia, but discovering them in an automated manner requires the consideration of Wikipedia's authoring culture. Aside from a technical challenge caused by the complexity of the grammar of the Wikimedia markup language [Vadim Zaytsev, 2011], its knowledge is managed by many contributors and as a result, is prone to inconsistencies in its quality [Stvilia et al., 2008]. To accurately detect relevant fragments of knowledge, understanding Wikipedia's peculiarities is crucial. In this thesis, we summarize and discuss the problems encountered when extracting knowledge from Wikipedia while also relating to DBpedia [Torres et al., 2012a, Torres et al., 2012b] as its semantically more structured mirror. We hypothesize that problems are manageable with the use of standardized techniques from data classification and machine learning. For each article we inspect its different sets of content features and provide statistics on them for all articles in a defined scope. To exemplify the sets of features and the problems encountered, we present bits of an exemplary article on the software MATLAB in Figure 1.1.

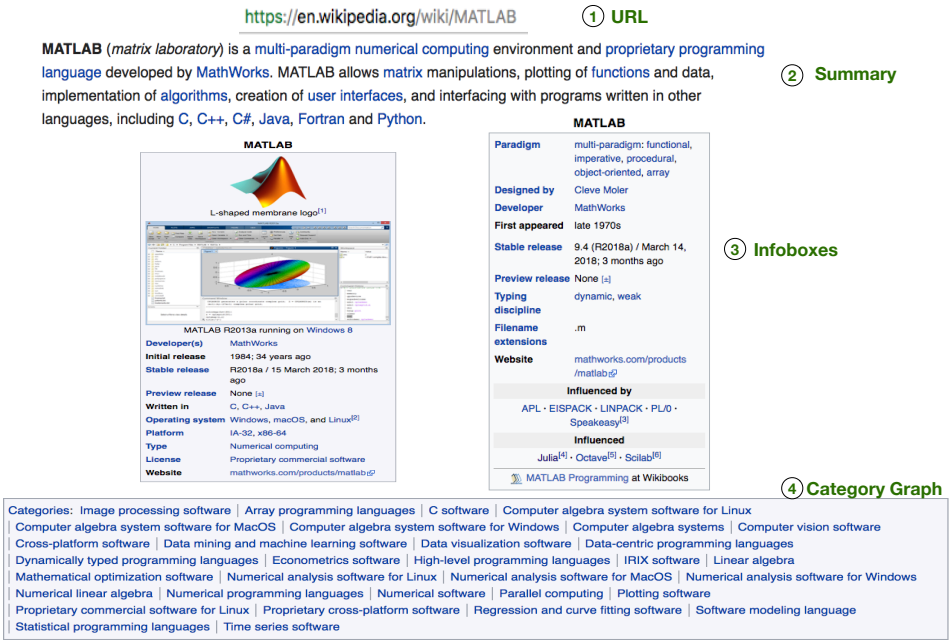

Figure 1.1 – General structure of a Wikipedia article and pointers at structural features.

The article on MATLAB[8] contains multiple content features. The URL repeats the title of the article. Essentially, the title corresponds to the name of the main topic, which can be interpreted as an ontological entity. The summary and in many cases specifically the first sentence provide a definition of the described entity. For MATLAB, the summary actually covers multiple entities. The language and the software implementing the language are both covered. In the infobox that uses the template for programming languages, we find typical properties of software languages, such as paradigms. Categories annotated to the article may have an intuitive value for classifying the software or the language, but one has to first differentiate between those referring to the language and those referring to the software.

We aim to recover a corpus of articles relevant to the ontological class software language. Such a corpus can later be reused to identify subclasses. While articles on domains such as animals [Wang et al., 2010] often contain scientific classification in infoboxes and can thus be easily reused, we found that articles on software languages are not easily recognizable by automated procedures. Two kinds of training data drive our efforts in the machine learning-

[8] `https://en.wikipedia.org/wiki/MATLAB` — Requested March 31, 2022

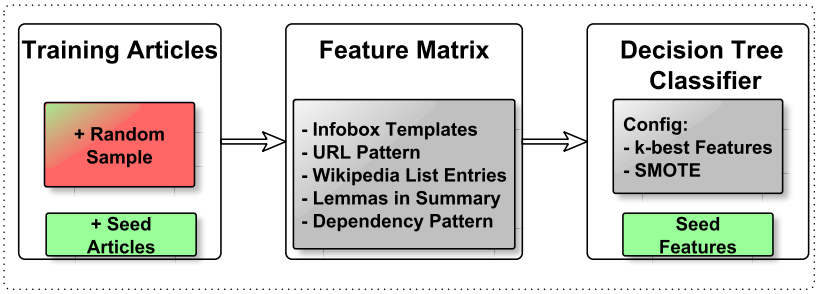

Figure 1.2 – An overview of the seed-based learning approach.

based methodology that is depicted in Figure 1.2. i.) We build a training set that includes a seed of representative relevant articles. ii.) Based on the seed, we identify the scope in which we intend to discover more articles. iii.) For all articles, we mine features with categorical values that state whether a content feature, for example, usage of the template 'infobox programming language', is present. iv.) With the training data, we train a binary classifier that decides whether an article is relevant. We evaluate our approach by a set of 990 articles that were labeled by experts in the software language engineering domain.

Thesis contribution: The contribution is published in [Heinz et al., 2019]. In Chapter 4, this thesis augments the published contribution as follows: i.) Features of articles that are used to classify them on their relevance are discussed in more detail. Respective pointers to Wikipedia's guidelines as well as literature are given for all features. Features that were not used but considered are covered as well. ii.) Challenges that need to be managed when recovering knowledge from Wikipedia articles are discussed more exhaustively. iii.) We present details of the survey that was conducted to recover relevance classifications by experts. Thereby, we discuss cases for which there still is a disagreement between raters. iv.) We determine what dimensions for classifying software languages can be identified in terms of Wikipedia categories that are aligned with existing literature.

1.3.3 Mining Technology Usage

Our goal is to gain insights into how a technology is used by analyzing open source software. For example, multiple pattern can be formulated that reflect common usage of EMF. Their detection is non-trivial, especially when references between different kinds of software artifacts need to be resolved. A generator model is encoded in XMI and needs to be adapted to configure the code generation process. It refers to both, the original Ecore model that it is based on and the Java packages that can be generated. By identifying the generator model, we may connect the three artifact types mentioned and even analyze their consistency on the level of code fragments. Figure 1.3 gives a first impression on the frequency of repositories using EMF with varying build systems. The repositories range from student projects that have been inactive for months to actively developed applications.

We perform a large-scale empirical analysis to detect the different types of models as well as consistency relations. To achieve this, we develop a methodology consisting of the following steps. i.) We formulate a catalog of patterns that we intend to measure in terms of frequency. ii.) We locate repositories using the technology. iii.) We select vanilla repositories so that measurement is not distorted by complex technology combinations or non-trivial identification of dependent components. iv.) We detect instances of the formulated patterns in the selected repositories. v.) We report the results. For the case study, we identify instances of patterns in 1438 repositories. The rule-based language QegaL that is used to implement

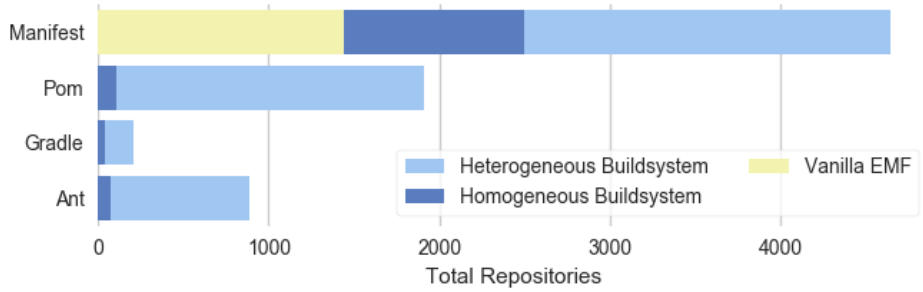

Figure 1.3 – Number of repositories with a particular build system further partitioned into homogeneous versus heterogeneous case.

the detection based on inference rules is at the center of the methodology. The rule engine works incrementally and stores results in a triple store.

> **Thesis contribution:** The contribution is published in [Härtel et al., 2018b]. In Chapter 5, this thesis summarizes the methodology and the results of the case study. QegaL in terms of the technical mining framework for pattern detection and the generic parts of the methodology to analyzing technology usage have been contributed by Johannes Härtel. The case study on EMF is a contribution in this thesis.

1.3.4 Modeling Technology Usage

Our goal is to construct models of technology usage that interconnect scattered knowledge resources that help with understanding technology usage, i.e., textual explanations and code examples. Such technology models are meant to complement existing reflections that, for example, focus on API method usage [Ratiu and Jürjens, 2007, Roover et al., 2013, Robillard et al., 2013]. To assist the reader with gaining a basic understanding of EMF conceptual overviews in terms of (fragments of) conceptual models are provided in [Garmendia et al., 2014, Ed-Douibi et al., 2016, Seybold et al., 2016, Troya et al., 2018]. Additionally, such conceptual models are also provided by referential literature, such as the book by Steinberg [Steinberg et al., 2008] that is dedicated to developers. An example of a technology model is presented in Figure 1.4. It includes central artifacts, such as an XMI-based data model, a configuration model (generator model), generated Java code, and their relationships when using EMF for Java code generation.

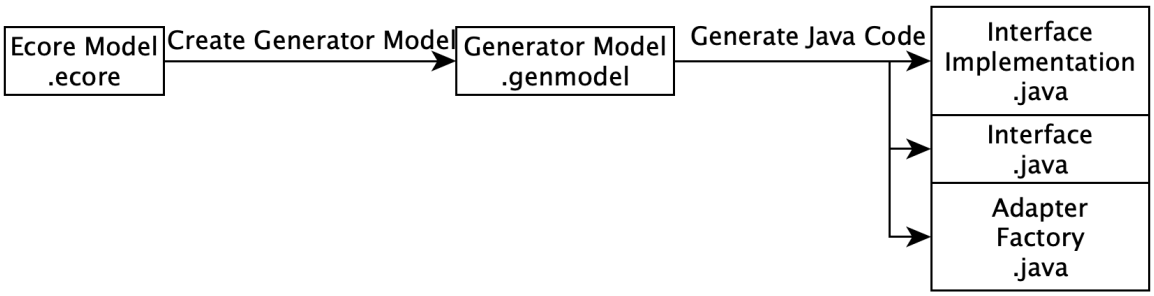

Figure 1.4 – A technology model on EMF-based code generation.

To identify evidence for artifacts and relations modeled in a technology model as in Figure 1.4, we contribute a methodology to systematically query literature and software projects. The methodology is independent of any specific technology for mining literature or software projects. Its central idea is the systematic reduction of selected corpora to linked

evidence. For example, the text in a book can be systematically reduced to relevant passages. The caption of a section often provides an indicator for different usage scenarios and can thus be reused to reduce the scope of a book to relevant (sub-) sections. To guarantee the reproducibility of a constructed technology model, we annotate links back to (sub-) sections or paragraphs from textual resources; to guarantee the reproducibility of the construction process, queries are shared as well. The methodology is evaluated by two case studies on EMF, where the first case study is meant to get the reader acquainted with a smooth execution of the methodology and the second demonstrates the robustness of the methodology against misconception.

Thesis contribution: The exploratory study in [Härtel et al., 2018b] serves as a basis for the research published in [Heinz et al., 2020], where the approach to mining repositories and projects is reused again. In Chapter 6, we present the contributions that are published in [Heinz et al., 2020].

1.4 Road map

Chapter 2 summarizes related research that serves as a foundation and inspiration. Hence, it also includes an overview on research previously conducted by the Softlang team.
Chapter 3 analyzes vocabulary on relationships between artifacts in the domain of megamodeling and presents a reference semantics for language-centric megamodels.
Chapter 4 discusses discovery of Wikipedia articles relevant to a domain class and a case study on software languages.
Chapter 5 describes the methodology and results of a large scale empirical analysis on EMF usage on GitHub.
Chapter 6 discusses the developed methodology to constructing technology models in a reproducible manner.
Chapter 7 summarizes contributions in related research contexts, in particular, related approaches in the domain of knowledge engineering, ontology engineering, (mega-) modeling, and empirical analysis of software language and technology usage.
Chapter 8 concludes with a summary and future work proposals.

Chapter 2

Background

In this thesis, we are particularly interested in recovering, maintaining, and evaluating knowledge resources from heterogeneous corpora. Therefore, we are inspired by common methodologies in existing research, for which we provide an overview. First, we provide an overview on existing research with a focus on the types of knowledge artifacts that we want to produce. Second, we discuss general aspects of evaluating knowledge that is dedicated to the different kinds of knowledge artifacts. Third, we discuss what resources can be used to discover knowledge on software languages and software technologies, namely, scientific literature, Wikipedia, developer literature and software projects. At last, we discuss previous research on software languages and technologies that directly inspired contributions in this thesis.

Chapter contribution: This chapter augments published contributions [Heinz et al., 2017, Heinz et al., 2019, Härtel et al., 2018b, Heinz et al., 2020]. We provide an exhaustive description of research that serves as a foundation. Hence, we focus on: i.) summarizing types of knowledge artifacts that we reuse or produce, ii.) discussing principles for quality assurance, iii.) presenting the state of the art of research on software languages and software technologies.

2.1 Knowledge Artifacts

The contributions of this thesis are inspired by research in the domains of knowledge engineering, ontology engineering, and megamodeling. Our research contributions include different types of knowledge artifacts that are well-known in scientific literature, namely, knowledge graphs [Hogan et al., 2020], ontologies [Corcho et al., 2006] and megamodels [Bézivin et al., 2004].

2.1.1 Knowledge Graphs

An extensive summary on research that is related to knowledge graphs is provided in [Hogan et al., 2020]. While we summarize the overall insights into knowledge graphs in [Hogan et al., 2020], we add references to exemplary work on Wikipedia and megamodels including the publications that this thesis is based on. A knowledge graph is any graph-like structure that encodes facts on a target domain, such as animals or software, in a structured manner, which makes it amenable to systematic and automated processing. It instantiates a schema, which is typically an ontology. A megamodel together with all of its instances can be considered as a knowledge graph as well, because the mentioned properties apply in the same way. Knowledge may be added in several ways. First, it may be manually added and maintained [Tanon et al., 2016]. Second, several automated mining tasks can reduce manual effort [Bizer et al., 2009, Morsey et al., 2012, Heinz et al., 2019]. Third, it may be deducted based on axioms that exist for an underlying ontology [Urbani et al., 2009, Heinz et al., 2017]. Fourth, it may be inducted based on analytical findings, i.e., commonalities in analyzed resources, where resulting inferences may be imprecise [Härtel et al., 2018b, Heinz et al., 2020]. Several large knowledge graphs exist, e.g., *DBpedia* [Bizer et al., 2009, Morsey et al., 2012].

Platform Goals

In existing knowledge graphs, cultural aspects need to be understood before the encoded knowledge can be properly reused. Tanon et al. [Tanon et al., 2016] discuss the human efforts that were necessary to migrate data from Google's knowledge base *Freebase* into *Wikidata*. They highlight partly manually resolved barriers that come with design decisions in available large knowledge graphs. One of the major barriers was a cultural difference in terms of the platforms' goals. While the *Wikidata* community emphasizes the necessity for references to reliable resources on every stated fact, not every fact in *Freebase* was linked to reliable resource. To resolve the issue, facts from *Freebase* were often treated as claims until a reliable resource was identified. In our research, we identified similar challenges for Wikipedia that are buried in author guidelines.

2.1.2 Ontologies

Our work adopts the notion of ontology engineering to structure knowledge on software as motivated in [Corcho et al., 2006]. We also adopt general principles to share knowledge, such as clarity and coherence, defined in [Gruber, 1995]. Multiple levels of knowledge can be distinguished. At the highest level of abstraction, there are so called general ontologies, such as DOLCE [Gangemi et al., 2002] or Cyc [Strom, 2000]. They define domain independent classes and relations as parts of a schema, e.g., part-hood. At the next level, Core Ontologies provide a formal basis for the core vocabulary of a domain [Scherp et al., 2011]. Then, domain ontologies model knowledge in a more specific domain, where multiple ontologies can exist for one domain [Corcho et al., 2006]. When different domain ontologies exist for the same domain, they are tailored towards different competencies.

The main steps for ontology development are requirement specification, refinement, and evaluation [Sure et al., 2002]. In the first step, the requirement specification defines the

purpose, pragmatics, and goal of the ontology. The intermediate result is a premature and semi-formal description that is driven by *competency questions*. A competency question states what the instances of classes and relations are used for. For example, "What are all the events happening in Santiago?" is the competency question for an ontology that consists of the classes *Event* and *Location* as well as the relation *locatedIn*, which express where events take place [Hogan et al., 2020]. Another example is given on software engineering in [Ren et al., 2014]: "Which processes implement an algorithm?". Hence, the ontology needs to cover the classes *Process* and *Algorithm* as well as the relation *implements*. In the second step, an ontology is refined through extension and adaption. The result is a formal description written in an appropriate representation language that has matured. Then, in the third step, the matured ontology is evaluated against the requirement specification and more general evaluation criteria, such as its integrity and usefulness.

In this thesis, we contribute to the domains of knowledge engineering as well as model-driven engineering at the same time, where ontology engineering can be seen as a part of knowledge engineering. Ontologies resemble metamodels and knowledge graphs resemble models [Bézivin, 1998, Aßmann et al., 2006, Staab et al., 2010]. In an early attempt to connect the research fields of model-driven engineering (*MDE*) and ontology engineering, a mapping of *MDE* terms to ontology engineering terms is explained in [Bézivin, 1998]. For example, in the introduction, the author clarifies that a metamodel resembles an ontology 'Models are defined (constrained) by meta-models that we will call here ontologies.' Further conceptual commonalities and differences between ontology engineering and *MDE* are discussed in [Aßmann et al., 2006, Staab et al., 2010].

2.1.3 Megamodels

Megamodels consider models and their model fragments as first-class entities and capture their relationships. Bézivin et al. report on the notion of megamodels in [Bézivin et al., 2004]. Three essential examples are discussed. i.) The first exemplary megamodel captures relations between product models and process models. Both types of models contain fragments that represent types of artifacts. The artifact types are defined by the product models, and used or produced by actors in the process models. UML class diagrams and generated Java code exemplify types of artifacts. ii.) The second megamodel considers the relation between schema and instance, where an instance model *conforms to* a metamodel, i.e., a schema. iii.) The last example is concerned with the traceability of transformations. Here, a megamodel interconnects three models at the instance level, namely, an input model, an output model, and a transformation model. Each of the three models conforms to a metamodel. The motives of the summarized examples appear again in our research in Chapter 3 and Chapter 6.

Five different types of megamodels are described in [Bagge and Zaytsev, 2014]. *Informal Megamodels* focus on understanding and communication [Bézivin et al., 2004]. Hence, they are encoded as visual diagrams with varying and informal vocabulary. *Ad hoc megamodels* as in [Zaytsev and Bagge, 2014] are on an architectural detail level. They are more detailed than informal megamodels. They depict which kind of concepts are related based on common vocabulary from a domain, such as parsing [Zaytsev, 2014] or model-driven engineering [Favre, 2005b, Favre, 2005a]. *Instrumental megamodels*, e.g. in [Härtel et al., 2017], are elements of a particular megamodeling language, such as *MegaL*. Before an author begins to construct a model, the names of relations and node types are fixed in terms of a modeling language. Megamodels have been put to use in various domains, such as DSL tools [Jouault et al., 2010], self-adaptive systems [Vogel and Giese, 2012], or type-checked model transformations [Vignaga et al., 2013]. The queries in Chapter 5 can be modeled as an instrumental megamodel used for measurements. *Formal megamodels* are based on a formal theory, with which researchers can verify megamodels through formal analysis [Favre, 2005c]. Hence, megamodels formulate formal properties and logical constraints, which resemble formal ontologies. We relate to

this type of megamodel in Chapter 3. *Space megamodels* model pattern in a domain, such as technology usage. The technology models as presented in Chapter 6 are encoded in such terms. They explain which central fragments are combined to form a pattern. *In this thesis, we specifically cover instrumental, formal as well as space megamodels.*

2.2 Quality Assurance

In this section, we discuss the challenges arising from crowd-efforts and their influence on quality. We also present approaches towards evaluating quality. Wikipedia and related resources, e.g., Wikidata, serve as examples.

2.2.1 Trustworthiness

For all of our contributions, we are particularly interested in high trustworthiness. In [Färber, 2019], trustworthiness is discussed as a quality attribute for facts in knowledge graphs. Trustworthiness metrics are aggregated over four subdimensions. A fact is seen as *believable*, when it is accepted without verification; it has a *high reputation*, when it is highly regarded in terms of its source or content; it is seen as *objective*, when it is unbiased and impartial; it is *verifiable* when it can be checked for correctness. Trustworthiness can be measured on different structural levels, where the measurement is based on manually defined criteria. On the level of a whole knowledge graph, trustworthiness is higher, when facts are curated by experts, and is lower, when facts are recovered from unstructured sources automatically without any kind of supervision. On the level of statements, trustworthiness is higher, when every property is traceable, and is lower, when no provenance is given.

2.2.2 Wikipedia Quality Evaluation

While inspecting non-expert resources, we aim at knowing cultural challenges of an original resource. Early doubts on Wikipedia articles' quality were addressed early by an article that compared Wikipedia to the more established *Encyclopaedia Britannica* [Giles, 2005]. The results show that the quality of Wikipedia was 'reasonably good'. An investigation by experts in 2005 found that articles in the *Encyclopaedia Britannica* had about three inaccuracies per article where Wikipedia had about four. Further involvement of scientists in the collaborative efforts are said to be the most beneficial to an encyclopedia that already has such a small barrier to accessing and contributing content.

The quality of Wikipedia articles is assured by appointed users that are organized in a hierarchy. Stvilia et al. [Stvilia et al., 2008] provide insights into Wikipedia's account organization. The study reveals how members play different roles in the overall contribution and maintenance processes. We found that the information in [Stvilia et al., 2008] is by now outdated concerning described permissions for account types such as *bureaucrats*. By now, official documentation for 19 different types of accounts ordered by the level of access is given on official pages by the Wikimedia foundation.[1] We describe exemplary types. *Flooders* are individual editors with ability to perform mass changes that are marked as bot activities. *Administrators* are allowed to edit, delete and undelete pages in most of the public Wikimedia wikis. *Bureaucrats* have permissions for managing administrator accounts. They can promote candidates to administrators or bureaucrats. Candidates are nominated through dedicated request processes. Administrators can later be demoted on request, when they violate policies or norms. In Section 4.1, we more extensively discuss Wikipedia as a valuable but challenging corpus.

[1] `https://meta.wikimedia.org/wiki/User_groups` — Requested March 31, 2022

2.2.3 Ontology Evaluation

After an ontology is constructed, it has to be evaluated against different quality criteria, such as accuracy or clarity. Different approaches to ontology evaluation exist that require different types of resources as a prerequisite [Raad and Cruz, 2015]. Table 1 in [Raad and Cruz, 2015] assesses how the four main evaluation methods, which are presented next, address quality attributes such as accuracy or clarity.

Gold Standard Evaluation

Gold standard evaluation compares a constructed model against a reference model. It is discussed in more detail in [Dellschaft and Staab, 2006] and in [Sfar et al., 2016]. A gold standard ontology is required in advance as the reference model. Thus, there needs to be an available ontology that the constructed ontology can then be compared against, e.g., by analyzing similarity [Maedche and Staab, 2002, Cheatham and Hitzler, 2013]. Identifying a suitable gold standard ontology is non-trivial, because the requirements and thus the goals (and hence the competencies) of the reference ontology and of the constructed ontology should match to a large degree [Cheatham and Hitzler, 2013, Raad and Cruz, 2015]. Otherwise, differences are an obvious result. This problem is more frequently used in the domain of genealogy [Ashburner et al., 2000], where multiple gene databases exist and differences need to be dealt with to reuse facts from one gene database in another.

Corpus-based Evaluation

A corpus-based evaluation compares the ontology against a representative corpus, such as a text corpus. It is also known as data-driven evaluation and is discussed in more detail in [Brewster et al., 2004, Hlomani and Stacey, 2014]. A representative corpus is selected for the target domain of the ontology as input. Typically, central terms are extracted from the corpus and then checked on whether they are covered by the ontology. This way, the ontology is compared with the referential corpus to especially assess its accuracy, completeness and conciseness.

Task-based Evaluation

With a task-based evaluation, an ontology is evaluated by using it to execute a predefined task. Hence, the ontology is evaluated based on a concrete application scenario [Raad and Cruz, 2015]. Thereby, the suitability for the stated competencies can be measured. An exemplary evaluation of this type is conducted on Wikipedia and its categories in [Yu et al., 2007], where the category graph is assumed to be a simple ontology and then evaluated. Categories are supposed to help with structuring the set of articles based on their topics. Hence, *browsability* is assumed as a competency. Users have to be able to explore a given domain. The experiment was conducted in the domain of racing sports as well as foods. More specifically, the tasks relate to competency questions such as “Which articles cover international racing competitions?” The efficiency and effectiveness of users to find relevant articles is measured for the original as well as a manipulated category tree.

Criteria-based Evaluation

Criteria-based evaluation assesses the quality of an ontology based on measures, e.g., complexity, consistency, conciseness, expandability and sensitivity [Raad and Cruz, 2015]. Inspired by existing metrics from software engineering, in [Navarro et al., 2010], a survey is conducted on ontology metrics that measure the complexity of an ontology related to graph and class metrics. Cohesion and coupling have been specifically discussed in [Kumar et al., 2017]. Bad smells

and their resolution by refactorings as well as design anomalies are covered in [Baumeister and Seipel, 2006, Baumeister and Seipel, 2010].

2.2.4 Conceptual Model Evaluation

Conceptual models are created to communicate complex intertwined aspects [Ben-Ari and Yeshno, 2006]. In the domain of conceptual models, quality evaluation approaches aim to cover largely the same quality attributes as for ontologies. In [Qi et al., 2010], a summary is provided in terms of an ontology of conceptual model quality attributes. An overview is presented in the lists below.

- *Syntactic:* Lucidity, soundness, laconicity.
- *Semantic:* Completeness, inherence, clarity, consistency, orthogonality, descriptive quality, minimality, correctness.
- *Pragmatic* Simplicity, understandability, inferential quality.
- *Social:* Agreeableness, timeliness.
- *Cognitive:* Perceived semantic quality, perceived ease of understanding, perceived usefulness, perceived evaluability, perceived comprehensibility, user satisfaction.
- *Usage:* Flexibility, integration, maintainability, reusability, reliability, implementability.

2.3 Target Domains

According to [Smith and Welty, 2001], databases and information systems, software engineering (in particular, domain engineering), and artificial intelligence create a demand for the application of ontology engineering. In this thesis, we are concerned with the domains of software languages and software technologies as they are central to any software engineering process.

2.3.1 Software Languages

Many attempts on classifying languages used for engineering software can be identified in literature. One of the earliest attempts to classify programming languages was made in Floyd's Turing award lecture [Floyd, 1979], in which the notion of paradigms is introduced as a pattern-like way of thinking. A guide to choosing paradigms and a programming language that supports them based on a problem description is offered in [van Roy, 2009] . Aside from programming languages, domain-specific languages [Walter et al., 2009] and architecture description languages [Medvidovic and Taylor, 2000] are covered in dedicated research. Classifying computer languages is more complex and cannot be solved by considering paradigms alone [Shilov et al., 2012]. Properties such as flexibility and naturalness of syntax should also be considered. In [Shilov et al., 2012], the creation of a collaboratively maintained flexible computer language ontology as well as a platform is proposed.[2]

The term software language is at the center of attention in the emerging 'Software Language Engineering Body of Knowledge (SLEBOK)' [Combemale et al., 2018]. This community effort aims at sharing knowledge on conceptual aspects in the field of software language engineering. *At the SLEBOK Dagstuhl seminar, a questionnaire was distributed, where the results disclose differences in researchers' understanding of the term software language.* The participants of

[2]To the best of our knowledge, the prototype platform does not exist anymore since provided links do not work and the mentioned 'Computer language classification knowledge portal' cannot be found.

the seminar were asked to state whether a given concept, e.g., 'UML diagrams' or 'Java', instantiates the notion of a *software language*. The results provide a detailed discussion on the term 'software language' including insights into disagreements in the software language engineering community. Four categories towards better understanding software languages were identified:

- *Agreed Positive:* domain-specific, programming, description, meta-, and modeling languages.
- *Agreed Negative:* artificial human, natural, and physics languages.
- *Context-sensitive:* API, constrained strings, Forms, UI, and spreadsheets.
- *No Consensus:* storage formats, encodings, structured text mechanical, and ontology languages.

Competencies of the term software language *are explicitly discussed in [Favre et al., 2009, Lämmel, 2018] and further stated in Chapter 3 for the ontological class.* In [Favre et al., 2009] a rationale on the creation of the term *software language* is provided, which we summarize as follows. Software is a conglomerate of programs, data files, models, etc. It is not just a synonym for a program. There exist many languages in the context of software development. We repeat a list of types of software languages: *'rule-based languages, formal specification languages, configuration languages, meta-languages, query languages, model-transformation languages, schema definition languages, requirement specification languages, domain-specific languages, protocol definition languages, scripting languages, text formatting languages, business-process description languages, architectural description languages, markup languages, modeling languages, etc.'* A commonly known term is that of a 'Computer language'. It is used on Wikipedia as well[3], but in [Favre et al., 2009], the authors state that this term is not well in line with modern software engineering, which is no longer only about communicating with machines. Instead, the focus shifts to the communication with human beings as well as deeply understanding programs that are perhaps executed by machines. This perspective raises awareness of measurable quality attributes of software languages such as conciseness and understandability.

At last, this perspective on the term software language motivates Favre et al.'s introduction of the term *linguistic architecture* in [Favre et al., 2012c], which addresses the need for linguistic studies in a software projects. In [Lämmel, 2018] the term software language is further refined in terms of properties and several classification dimensions such as a purpose-based classification. Any software language is defined by a grammar, well-formedness rules, semantics and pragmatics. Such statements can be interpreted as competencies of the ontological class software language as discussed in Section 3.2.

2.3.2 Software Technologies

Research on software technologies that are used by software engineers, e.g., on frameworks or libraries, is typically revolving around programs. We summarize existing literature in the main directions of program comprehension and API usage, where our research can be seen as complementary.

Understanding API Usage

In [Robillard et al., 2013], a survey on knowledge engineering techniques in literature that aim at understanding properties of APIs is given. The focus is on general properties of an API

[3] `https://en.wikipedia.org/wiki/Category:Computer_languages` — Requested March 31, 2022

that influence the software developed based on the API. Thus, browsing or searching API components (e.g., [Lv et al., 2015]), code-example retrieval (e.g., [Dagenais and Robillard, 2012]), design recovery of the API itself (e.g., [Ellis et al., 2007]), and software metrics for APIs (e.g., [de Souza and Bentolila, 2009, Sobernig et al., 2011]) are excluded. The authors provide an overview on various categories of API property research, namely, i.) Unordered API usage pattern that discusses which API elements coexist frequently (typically based on association rule mining); ii.) Sequential usage pattern that take order of API elements into account through a largely varying set of mining techniques; iii.) Behavioral specifications that consider reachable system states (e.g., through symbol execution); iv.) migration mappings; and finally v.) general information that deals with more idiosyncratic properties (e.g., aiming at measuring popularity).

Several empirical studies aim to identify what obstacles exist to understanding software technology. In [Nykaza et al., 2002], the authors collect answers by programmers on what they expect from learning resources and environments that are supposed to assist with understanding software development kits. In [Robillard, 2009], textual explanations and code examples are identified as essential resources for understanding API (method) usage. This insight results from surveying and interviewing Microsoft developers. In [Duala-Ekoko and Robillard, 2012], the setup and results of a controlled experiment on understanding APIs with student participants is presented. Several observations are reported, from which we focus on the following two: i.) Discovering the relevant API types that may not be directly connected by a reference is a major challenge; ii.) Using the names of API elements as part of the search string for browsing the web and documentation is effective for finding code examples.

Identifying what part of an API is used in an exemplary code fragment that was ripped out of context is difficult. In [Dagenais and Robillard, 2012], an approach to detect code fragments in a largely natural language document that are then to API methods is proposed. In [Subramanian et al., 2014], an improved linking mechanism is presented that can reliably identify fully qualified names of API elements based on an oracle database. The proposed tool *Baker* can deal with JavaScript and Java-based programs. In [Uddin et al., 2012], the repository history is analyzed to identify common temporal progression in terms of API usage. API concepts are identified based on frequent item set mining. Thus, they consider common coexisting API methods as a concept. Then, they study common orders of introduction for such API concepts in projects.

Knowledge Engineering for APIs

Related work exists that tackles the problem of understanding an API through knowledge engineering approaches. In [Ratiu and Deissenboeck, 2006, Ratiu and Jürjens, 2007, Ratiu and Jürjens, 2008, Ratiu et al., 2008, Feilkas and Ratiu, 2008] research is presented on software technology that focuses on concepts in the reflected domain of an API, such as geometry. Programs are treated as knowledge graphs in [Ratiu and Deissenboeck, 2006]. Hence, the authors search for semantic relations based on the syntactic structure as well as the type system. Program elements like variable or class names are mapped to an ontology. This mapping is performed with the help of WordNet. In [Ratiu and Jürjens, 2007], the mapping for library code and an ontology again based on WordNet is discussed. In [Ratiu et al., 2008], domain ontologies are extracted from APIs that reflect the concepts being represented by reusable interfaces and classes. In [Ratiu and Jürjens, 2008], the authors propose to evaluate the quality, specifically understandability and usefulness, of an API by checking the coverage of a given domain ontology against concepts represented in an API. For example, a domain concept may be represented by a class or variable name that can be matched. Hence, the number of matched concepts can be used as a measure. In [Feilkas and Ratiu, 2008], the authors consider further syntactic constraints that are assumed when reusing library functions.

API Documentation

In a series of work, Robillard et al. further elaborate on research towards better understanding API documentation. In [Maalej and Robillard, 2013], the authors present a study on reference documentation for APIs on a meta-level. They contribute a taxonomy of types of knowledge encoded by an API documentation and discuss how these types are distributed. What kinds of information positively or negatively affect an API documentation is investigated in [Uddin and Robillard, 2015],. For this purpose, an exploratory survey is set up with 323 IBM software engineers, in which the negative attributes incompleteness, ambiguity, obsoleteness, incorrectness, inconsistency, and unexplained examples were named as typical blockers. Textual explanations relevant for a given API element are discovered in [Robillard and Chhetri, 2015]. The identification is based on word pattern. A word pattern is identified in a semi-automated manner by natural language processing, more concretely, part-of-speech tagging and chunking. For example, the words 'may' and 'efficient' co-appear frequently in documents. Hence, they serve as a pattern to identify a knowledge item. In [Petrosyan et al., 2015], tutorial sections in a documentation resource for a given API element name are discovered. First, candidate sections are identified by matching the name. Second, the candidates are classified on their relevance based on text features. To reduce the dimension of text features, mentioned API elements are collapsed to a single keyword. Textual features correspond to several heuristics, such as how frequently the term is mentioned within a section. In similar manner, relevant Stackoverflow fragments are recovered and then inspected in [Treude and Robillard, 2017].

Software Technologies on Wikipedia

The tool Witt is proposed in [Nassif et al., 2019] for automatically categorizing software technologies based on Stack Overflow and Wikipedia. Tags are extracted as concepts from Stack Overflow and matched with Wikipedia articles. They are assigned to a technology, if a hypernym relation can be recovered between the technology and the tag in the Wikipedia article. In [Nassif et al., 2020], the authors further elaborate on the approach and further involve article links, infobox values, Wikipedia's categories, and extended natural language processing.

2.4 Models of Software Language and Technology Usage

Summarizing previous work, the terminology of linguistic studies in the field of software engineering gives credit to the multitude of languages involved in the usage of complex software technology [Favre et al., 2012c]. The studies are no longer restricted to analyzing and reporting findings in programs that are part of a software, but also respect other types of artifacts, such as models and grammars, which are element of a software language. In this section, we present the research that most directly inspired contributions in this thesis.

A literature survey is initiated in [Favre et al., 2011] that focuses on what aspects are investigated by linguistic studies of software. For each surveyed paper, the effort provides insights on the objectives, analysis type, and the used software corpus. Characteristics of the corpora are collect with a focus on conceptual roles (called *units*) and predominant languages.

We have been contributing to the software chrestomathy '101companies', which is a collection of small software systems that implement a common feature model while aiming at representing best practices for a variety of language usage, technology usage, and software design [Favre et al., 2012b]. The implemented data model is presented in Figure 2.1. The systems are documented on a semantic wiki; the documentation includes properties of language and technology usage. Furthermore, the term software chrestomathy, especially, in comparison to a program chrestomathy is discussed in [Lämmel, 2015]. By implementing the same features,

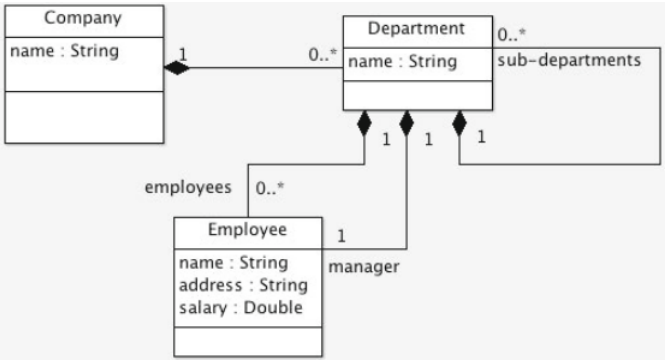

Figure 2.1 – A simple data model of a company, where the operations of cut and total are implement using varying software languages and software technologies. Reprinted by permission from Springer Nature, "Objects, Models, Components, Patterns" [Favre et al., 2012b], Copyright © 2012, Springer-Verlag Berlin Heidelberg.

technologies and languages can be more directly compared. The focus of studies based on contributions to a software chrestomathy is no longer only on program artifacts, but also on other kinds of data, such as XML files, and its role within an executable system, i.e., the heart of a chrestomathy contribution that is then documented.

Figure 2.2 – Browsing facilities for contribution to the 101companies project are introduced in [Favre et al., 2012a]. Users can explore software languages and software technologies used in contributions. The figure is reprinted from [Favre et al., 2012a] © 2012 IEEE.

Based on contributions to the *101companies* project, the ecosystem for understanding the used software languages and software technologies is introduced in [Favre et al., 2012a]. Every artifact is classified by the used language and technology as well as the realized concepts. An example for a project overview that allows browsing through projects or the used languages, technologies and concepts is depicted in Figure 2.2. In [Lämmel et al., 2014a], the contributions are compared to each other with regard to quality attributes. For each implemented feature, the lines of code needed by a contribution is measured.

In [Schauss et al., 2017], a chrestomathy for DSL implementation is presented, where idioms for DSL implementations are demonstrated using different technologies and languages. The used technologies include *ANTLR* (Java and Python), *EMF*, and *Xtext*. Concepts common for implementing a DSL are systematically recovered by filtering frequent (TF-IDF) terms from the reference documentation for the technology. The contributions are explorable in a web documentation similar to Figure 2.2. Conceptual overviews that classify contributions also link to 101wiki, which has already been used for previous chrestomathies.

A chrestomathy focusing on contributions written in and exemplifying Haskell is presented

in [Lämmel et al., 2013c]. Knowledge is integrated from several literature resources, for example, the web resources *HaskellWiki* and *Wikipedia* and the books 'Learn You a Haskell' and 'Real World Haskell'. The textbooks are processed to assess and improve coverage of links from the maintained vocabulary across the semantic wiki *101wiki*. Figure 2.3 summarizes what is linked by a *101wiki* page.

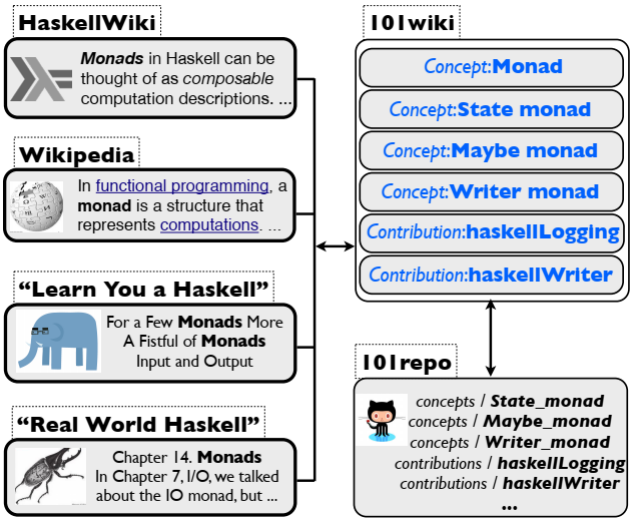

Figure 2.3 – Resources reused, processed and systematically linked for a chrestomathy focusing on Haskell contributions [Lämmel et al., 2013c]. Here, artifacts are typed by a language instead of metamodels. Republished with permission of ACM, from [Lämmel et al., 2013c] © 2013 ACM; permission conveyed through Copyright Clearance Center, Inc.

In [Roover et al., 2013], support for empirical analysis of API usage is presented. A corpus containing matured software projects with resolved dependencies is delivered, which is deemed suitable for API usage analysis. An exemplary investigation into the usage of JDOM is conducted. The table in Figure 2.4 provides on overview on the frequency of usage for packages and classes that are part of JDOM, which exemplifies the explorative dimension of usage metrics. Hence, an API-centric perspective is provided on the corpus, while a project-centric perspective is also discussed. Furthermore, investigations show that an API has different facets, where a facet is a group of API elements that appear together but are typically separated from other groups of API elements. Hence, one API serves for different scenarios in software projects. The paper also elaborates on how multiple APIs are frequently used together. The coupled APIs are typically part of the same programming domain.

Pattern	#projs	#refs	#elems	#derives
org.jdom	6	2391	84	5
Element	5	1912	44	1
Document	5	160	6	1
Namespace	4	82	6	0
Attribute	4	70	6	0
Text	4	67	4	2
JDOMException	6	54	4	0
Content	3	21	6	0
CDATA	3	9	1	0

Figure 2.4 – An API-centric perspective on the corpus. Here, frequency of API element usage is presented as a table. *#projs* is the number of projects, *#refs* is the number of references over all projects, *#elems* is the number of API elements used, *#derives* is the number of project types derived from API types. The table is reprinted from [Roover et al., 2013], © 2013 IEEE.

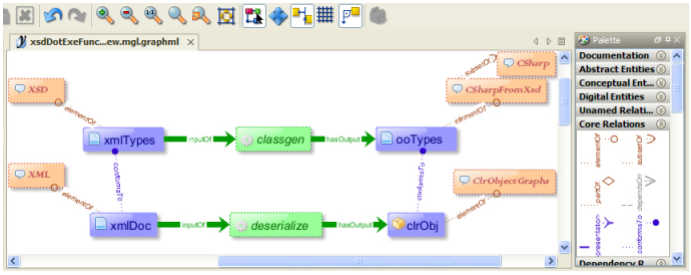

Figure 2.5 – An initial megamodel of artifacts in an O/X mapping scenario copied from [Favre et al., 2012c]. Here, artifacts are typed by a language instead of metamodels. Reprinted by permission from Springer Nature, "Model Driven Engineering Languages and Systems" [Favre et al., 2012c], Copyright © 2012, Springer-Verlag Berlin Heidelberg.

Multiple modeling languages tailored towards studying software languages and software technologies in software projects have been designed [Lämmel et al., 2014b], most notably MegaL[4] [Favre et al., 2012c]. A MegaL model declares how artifacts (such as files or objects) and concepts (such as languages or programming techniques) relate in the context of scenarios of software technology usage. In Figure 2.5, O/X mapping is modeled in the way it is supported by the software technology JAXB. Artifacts are in the center of attention (*xmlTypes*, *ooTypes*, *xmlDoc*, *clrObj*). They are classified by their language, related by conformance, and related to functions (i.e., operations) like code generation (*classgen*) and deserialization (*deserialize*). A more detailed analysis of languages is suggested in terms of a *subset* relation. The artifact *ooTypes* is not only an element of C-Sharp, but more specifically, it is an instance of the kind of C-Sharp that can be generated from an XSD file by JAXB. Such megamodels can be 'renarrated' [Zaytsev, 2012, Lämmel and Zaytsev, 2013, Zaytsev, 2014], i.e., they can be presented in a story-telling manner. This way, the encoded information is explained by a further descriptive text that aims to improve understandability.

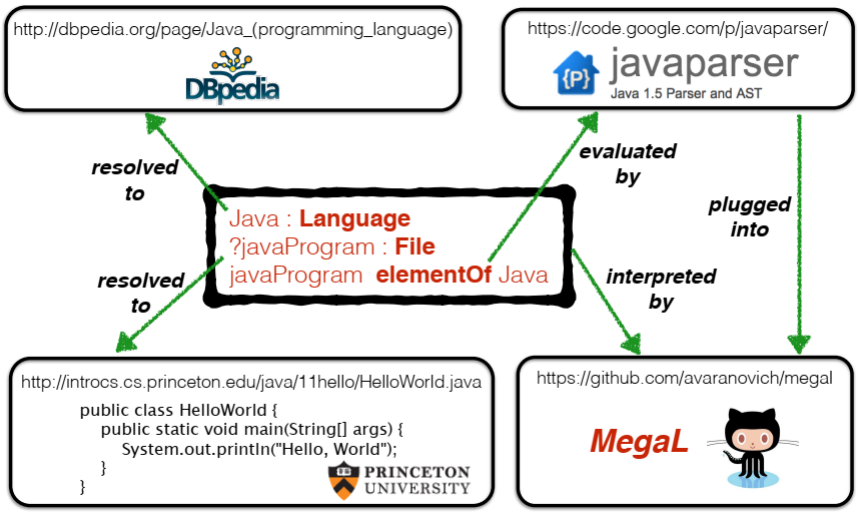

Figure 2.6 – A conceptual overview from [Lämmel and Varanovich, 2014] on interlinking various knowledge resources including concrete software artifacts, knowledge graphs to enable a plugin-based verification of MegaL statements. Reprinted by permission from Springer Nature, "Modelling Foundations and Applications" [Lämmel and Varanovich, 2014], Copyright © 2014, Springer International Publishing Switzerland.

A verification approach for various MegaL relations is proposed in [Lämmel and Varanovich, 2014]. It is based on linking a megamodel entity to a concrete artifact. For this megamodel

[4] `http://www.softlang.org/megal` — Requested March 31, 2022

entity, a relationship may be stated. The relationship itself together with the target of the relationship is then mapped to a program. Such declarations can be verified [Lämmel and Varanovich, 2014]. This allows checking whether a MegaL statement is verifiable as a form of interpretation. Additionally, conceptual entities, such as a software language, may be linked to a knowledge resource like a DBpedia page for an ontological classification. Figure 2.6 summarizes what resources are interconnected by a verifiable megamodel.

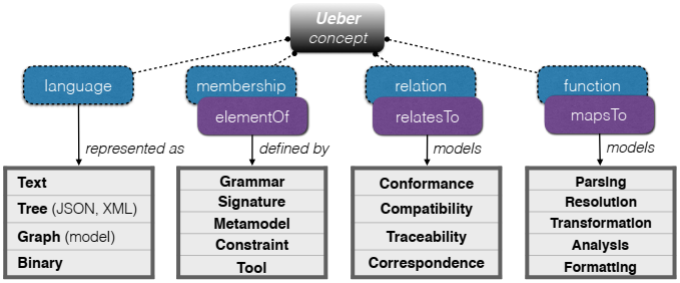

Figure 2.7 – An overview of what kind of information serves as summarizing documentation for artifacts in a demonstrative software project in [Lämmel, 2017]. The figure is reprinted from [Lämmel, 2017] and licensed under `https://creativecommons.org/licenses/by/4.0/`.

In [Lämmel, 2017], an approach to maintaining demonstrative software projects that exemplify concepts discussed in the book on software languages [Lämmel, 2018] is presented. The use of a heterogeneous set of software languages requires specific competencies of a megamodel for maintaining a consistent state. The competencies of *Ueber* models are summarized in Figure 2.7. Every language as classified by its representation type (text, tree, graph, or binary). Every artifact's membership to a language can be checked, because the artifact defining the language is registered as well (grammar, signature, metamodel, constraint, tool). Artifacts have a common megamodeling relationship (conformance, compatibility, traceability, correspondence). Artifacts may be used as input or output of common functions (parsing, resolution, transformation, analysis, formatting).

In [Rocco et al., 2017], megamodels are used for managing model repositories. The goal is to maintain consistency between models within a project by automatically responding to edit operations. A project contains model artifacts that are in a consistency relationship. For every two models in a consistency relationship, delta models are used to express the difference between them and are referenced by the megamodel as well as the models themselves. When a model or the delta is changed, consistency is recovered by recomputing one of the models or the delta. In [Rocco et al., 2017], an integration of the recovery process as an adaption of MDEForge [Basciani et al., 2014] is sketched as well.

In [Härtel et al., 2017], the competencies of MegaL models are more deeply explored. Tool support and conceptual principles for linguistic architecture models are introduced, where links between megamodel elements (entities and relations) and code are systematically supported for an analysis based on the alignment of a model with a system. Beyond the competencies in Figure 2.8 that can be partly automated, the abstraction from a system is defined in terms of a catalog of competency questions. The competency questions lead to a more formal discussion of types and relations within the scope of linguistic architecture as discussed in the next chapter.

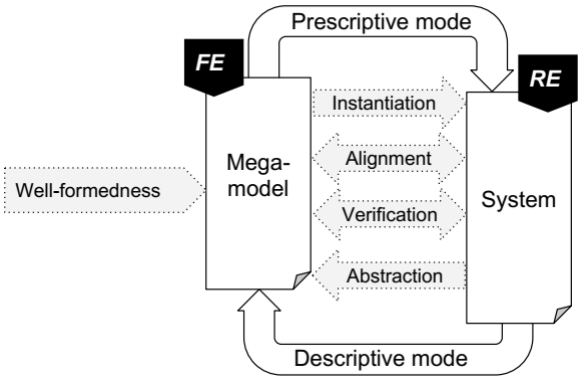

Figure 2.8 – A conceptual overview from [Härtel et al., 2017] elaborates on a megamodel-driven system engineering approach. The dimensions in which megamodels are clarified according to their usefulness are presented. The figure is reprinted from [Härtel et al., 2017], which is licensed under `https://creativecommons.org/licenses/by/4.0/`.

Chapter 3

Axioms of Linguistic Architecture

In documenting software technologies (e.g., web application frameworks) and specifically when discussing technology usage scenarios, software engineers aim to identify and classify the involved entities (e.g., languages and artifacts); they also describe relations between the entities (e.g., conformance or I/O behavior of program execution). Technical documentation provides a corpus of documentation idioms, where many facts are stated informally as part of natural language text or conceptual diagrams, e.g., in referential documentation of EMF or Django. In comparison, scientific literature provides a more formal basis, especially in the megamodeling domain. For example, recurring documentation idioms are 'a software *system* (e.g., a Python web application) to *use* a *technology* (e.g., the web application framework Django)' or 'a *model* (e.g., an Ecore model) to *correspond to* an *object model* (e.g., Java classes generated by EMF)'. We identify such documentation idioms, in particular the types of entities and relations used, in literature by means of a systematic literature survey on megamodeling. Then, we discuss a formal axiomatization of identified idioms for documenting software that is driven by megamodeling vocabulary. Thereby, we provide a formal reference semantics for the recurring documentation idioms, where the quality is assured by the construction based on established literature and examples drawn from model-driven engineering.

Chapter contribution: Most of the results discussed in this chapter are published in [Heinz et al., 2017]. The thesis augments the publication as follows: i.) The literature survey in [Heinz et al., 2017] that only includes papers until 2017 is updated with papers published between 2017 and 2020 in this chapter. ii.) The axioms are slightly modified to better separate laws on integrity and laws for inferring facts. iii.) We add competency questions to motivate groups of formal axioms. iv.) The axioms are implemented in Prolog for validation purposes.

3.1 Megamodeling Vocabulary in Literature

This section is dedicated to answering the research question on vocabulary in megamodeling literature:

RQ A1: What types of entities and relations are common in megamodels?

The results of the literature study motivate the types of entities and the relations that we formalize in this section. The accumulated occurrence of types across literature validates their practical importance. The executed methodology follows common steps for systematic mapping studies [Elberzhager et al., 2012], which is commonly used to recover overviews of a research domain. First, we define the scope in which we search for papers. Second, we define the search string. Third, we define additional inclusion and exclusion criteria to filter papers from search results. Fourth, while classifying the papers by the present types of megamodel entities and relations, we develop the classification scheme incrementally and discuss results.

3.1.1 Scope

We search for papers at 'ACM Digital Library' (ACM)[1], 'Springer Link' (Springer)[2] and 'IEEE Xplore Digital Library' (IEEE)[3] using the sites' search engines with the search string *"'mega model" OR "mega-model" OR "megamodel"'*. While ACM's and IEEE's default search settings only consider structured content (such as title, abstract and keywords), for Springer, we had to manually check search results for a match in the abstract, title or keywords while restricting the results to be in the software engineering category 'SWE'. We did not perform snowballing [Wohlin, 2014] to limit the amount of papers, as the analysis for paper inclusion and the classification is relatively laborious and automation is not trivially achievable. This way, we end up with 34 papers returned by our search efforts.

3.1.2 Exclusion

We screened the identified papers explicitly for relevance by the following criteria.

1. We *exclude* explicit *doubles* from the search results.
2. We *exclude* workshop summaries as the text rather introduces or summarizes addressed topics and *lacks a structured definition.*
3. We *exclude* papers that *lack a structured definition* of types of megamodel elements in a dedicated section, a schematic notation, or a metamodel.
4. We *exclude* papers that only show language elements that are presented in a *preceding publication.*
5. We *exclude* papers, when they are *preceding a publication* and the newer publication refines types and relations.

We discuss the justification for excluding specific papers from our initial search in terms of honorary mentions. The search returns two workshop summaries that we exclude, namely 'First International Workshop on Global Integrated Model Management' [Bézivin et al., 2006] and 'Introduction to GaMMa 2006 First International Workshop on Global Integrated Model Management' [Bézivin et al., 2006]. In the summaries, no structured definition of vocabulary is provided and thus they lack ground for classification. We found more papers,

[1] `http://dl.acm.org/` — Requested March 31, 2022
[2] `http://link.springer.com/` — Requested March 31, 2022
[3] `http://ieeexplore.ieee.org/Xplore/home.jsp` — Requested March 31, 2022

which *lack structured definitions* of types of entities and relations [Famelis and Chechik, 2019, Afzal et al., 2018, Vadim Zaytsev, 2017, Diskin et al., 2013, Sottet et al., 2009, Ceri et al., 2012, Perovich et al., 2009, Bézivin et al., 2005c, Chechik et al., 2016, Vogel et al., 2010, Ceri et al., 2013, Bastarrica et al., 2014, Heinrich, 2016, Hilliard et al., 2010]. In those papers, only informal explanations of examples are given, but no dedicated text explains or explicitly enumerates types of entities and relations.

Several papers reuse languages that have been introduced in *preceding publications*. We exclude them, because the types would simply be the same as in the original paper. The paper 'Interpretation of Linguistic Architecture' [Lämmel and Varanovich, 2014] succeeds [Favre et al., 2012c] by providing a deeper discussion on the interpretation of the language. Also, 'Renarrating Linguistic Architecture: A Case Study' [Zaytsev, 2012] summarizes elements from the language that are already presented in 'Modeling the Linguistic Architecture of Software Products' [Favre et al., 2012c]. Here, only the range of 'realisationOf' could be interpreted as function application, which is missing in the preceding paper. 'Model-Driven Engineering of Self-Adaptive Software with EUREMA' [Vogel and Giese, 2014] summarizes elements from the language that is already presented in 'A Language for Feedback Loops in Self-Adaptive Systems: Executable Runtime Megamodels' [Vogel and Giese, 2012]. In a similar way, 'Systematic Usage of Embedded Modelling Languages in Automated Model Transformation Chains' [Fritzsche et al., 2008] summarizes elements from the language that is presented in 'Model driven management of complex systems: Implementing the macroscope's vision' [Barbero et al., 2008]. The publication by Perdita Stevens [Stevens, 2018] also clarifies language constructs that to some degree have been sketched in previous work [Stevens, 2017], while the follow-up journal publication in [Stevens, Perdita, 2019] does not include any explicitly dedicated paragraphs for explaining types. Hence, we exclude [Stevens, 2017, Stevens, Perdita, 2019]. We also exclude 'Typing in Model Management' [Vignaga et al., 2009] as it precedes 'Inter-DSL Coordination Support by Combining Megamodeling and Model Weaving' [Jouault et al., 2010] which extends the defined megamodeling language.

3.1.3 Paper Classification

We classified included papers by the entity types and relations defined in them. For each paper, we extract explicit names of types of entities and relations. As a starting point, we use the vocabulary in [Favre et al., 2012c] to determine an initial set of types and relations. When we find types or relations that cannot be aligned with already recognized types and relations, we incrementally updated this set in analogy to updating the classification scheme by new keywords as part of a typical process towards a systematic map [Elberzhager et al., 2012]. As an obvious threat to validity, such an alignment procedure is based on our experience and expertise with the vocabulary from previous work. Thus, deep insights concerning the vocabulary introduced by other researchers may be lost. This threat could only be compensated through a laborious collaborative process. As a result, we formulate observations as objective as possible, but we cannot neglect the possibility of drawing wrong conclusions from inspecting the vocabulary and the given explanations in a paper.

Types of Entities

As the next intermediate result, we present the entity types that we identified. An overview is presented in Table 3.1, where the first column references the inspected paper and the second states which dedicated part of the paper was used for the recovery. Other columns present for which entity types there exists aligned vocabulary.

Artifact: In [Favre et al., 2012c], megamodel instances of the type *Artifact* represent digital software artifacts such as files, folders, or transients. Most of the papers in our survey do not use the type *Artifact* but relate to various subtypes instead. Every model, whether it is a

Paper		Entity Types									
		Artifact	*Function*	*System*	*Technology*	*Language*	*Fragment*	*Collection*	*Trace*	*Concept*	*Other*
[Favre et al., 2012c]	Subsections 3.2 and 3.3	x	x		x	x	x	x			
[Favre, 2004]	Figure 22	x		x	x					x	x
[Favre and Martinez, 2006]	Fig. 1	x	x								
[Gasevic et al., 2007]	Figure 8			x							x
[Barbero et al., 2008]	Figure 4	x						x	x		x
[Seibel et al., 2010]	Figure 8	x					x		x		x
[Jouault et al., 2010]	Figure 10	x							x		
[Kling et al., 2011]	Figure 2	x	x						x		
[Vogel and Giese, 2012]	Figure 10	x	x								x
[Méndez-Acuña et al., 2013]	Figure 3, Listing 1	x	x						x		x
[Vignaga et al., 2013]	Figure 1	x	x								
[Lämmel et al., 2014b]	Figure 2				x	x				x	x
[Simmonds et al., 2015]	Subsection III.A	x	x		x				x		x
[Salay et al., 2015]	Figure 1, Section II	x									
[Lämmel, 2016]	Figure 1	x	x			x					
[Toure et al., 2017]	Fig. 1, Subsection IV.B	x	x			x			x		x
[Stevens, 2018]	Section 3	x	x								
[Sottet et al., 2018]	Fig. 4	x					x				

Table 3.1 – Entity types in relevant papers discovered by the survey. The first line refers to the paper used for determining an initial set of vocabulary for common entity types. The last four lines contain the update, i.e., four papers that are not used in [Heinz et al., 2017].

metamodel or data model, is an artifact. Table 3.2 illustrates the many variations of artifacts in megamodeling literature.

Paper	Keywords
[Favre et al., 2012c]	Artifact, File, Program, ObjectGraph, Resource
[Vogel and Giese, 2012]	Megamodel, Model
[Simmonds et al., 2015]	Terminal Model, metamodel, metametamodel, megamodel
[Favre, 2004]	Metamodel, Model
[Favre and Martinez, 2006]	PIM Metamodel, PSM Metamodel, ISM Metamodel
[Seibel et al., 2010]	Model, DynamicHierarchicalMegaModel
[Salay et al., 2015]	Model, Transable, MegaTransable, Megamodel
[Jouault et al., 2010]	WeavingModel,TraceModel,TerminalModel
[Barbero et al., 2008]	Megamodel, Model
[Kling et al., 2011]	Model, ReferenceModel, TerminalModel, MetaMetaModel, Meta-Model, MegaModel
[Méndez-Acuña et al., 2013]	MetaMegaModel, MetaModel, Model
[Vignaga et al., 2013]	Model, TextualEntity, TerminalModel, MegaModel, Reference-Model, MetametaModel, MetaModel
[Stevens, 2018]	Meta-Model, Model, Safety, Code, Tests
[Lämmel, 2016]	Artifact
[Sottet et al., 2018]	Model, Metamodel, Megamodel
[Toure et al., 2017]	Model, Metamodel, Megamodel

Table 3.2 – Vocabulary aligned with the type *Artifact*.

Fragment: Only a few entity types relate to a part of an artifact, i.e., a *fragment* (see Table 3.3). We find the term "model element" to be used when megamodels are created to communicate MDE-related insights. It can be aligned with the more general type *fragment* which is used in the software engineering domain for parts of programs and models.

Paper	Keywords
[Favre et al., 2012c]	Fragment
[Seibel et al., 2010]	ModelElement
[Sottet et al., 2018]	Atomic Model element

Table 3.3 – Vocabulary aligned with the type *Fragment*.

Language: In most papers included by the survey, the type *language* is not explicitly considered. As observed in Table 3.4, previous work that is directly related to this thesis explicitly includes the type *language*. In such papers, a software language is perceived as a conceptual entity that represents a set of artifacts that conform to the language's definition of syntax, type system, semantics and pragmatics. Only the paper by Toure et al. [Toure et al., 2017] enumerates a set of specific languages for encoding metamodels.

Paper	Keywords
[Favre et al., 2012c]	Language
[Lämmel et al., 2014b]	Language
[Lämmel, 2016]	Language
[Toure et al., 2017]	UML, RDBMS, KM3, Java, QVT

Table 3.4 – Vocabulary aligned with the type *Language*.

Function: In megamodeling literature, we find entities that relate to operations and transformations (see Table 3.5). As a mathematical representation, we relate to this type of entities as a function.

Paper	Keywords
[Favre et al., 2012c]	Function
[Vogel and Giese, 2012]	OperationBehavior, ModelOperation, MegamodelCall
[Simmonds et al., 2015]	Transformation model, injector
[Favre and Martinez, 2006]	Refinement PIM-PSM, Refinement PSM-ISM
[Kling et al., 2011]	TransformationModel
[Méndez-Acuña et al., 2013]	M2MTransformation
[Vignaga et al., 2013]	TransformationModel
[Lämmel, 2016]	Function
[Sottet et al., 2018]	Mapping, Transformation
[Toure et al., 2017]	Global Operation

Table 3.5 – Vocabulary aligned with the type *Function*.

System: Three papers are referenced in Table 3.6, where a software system is represented by an entity as part of a megamodel. To document technology, megamodels in [Favre et al., 2012c] describe a system entity (comparable to a component) together with its relationships to programs and artifacts that are part of it.

Paper	Keywords
[Favre, 2004]	AppliwareItem
[Gasevic et al., 2007]	System, PhysicalSystem, DigitalSystem, AbstractSystem
[Favre et al., 2012c]	Program, Library, Technology

Table 3.6 – Vocabulary aligned with the type *System*.

Technology: We refer to technologies as reusable software components that software systems can rely on, such as libraries and frameworks. In Table 3.7, we additionally find the terms

'metaware' and 'engine' as hints towards this group in papers. The exact term 'Technology' from [Favre et al., 2012c] is used again in [Lämmel et al., 2014b] to represent reusable software systems.

Paper	Keywords
[Favre et al., 2012c]	Technology, Library
[Simmonds et al., 2015]	Transformation Engine
[Favre, 2004]	MetawareItem
[Lämmel et al., 2014b]	Technology

Table 3.7 – Vocabulary aligned with the type *Technology*.

Collection: In literature, we identify vocabulary that relates *collections* as a form of entity, where structured groups of entities are discussed. The identified vocabulary is given in Table 3.8.

Paper	Keywords
[Favre et al., 2012c]	Set, Pair
[Barbero et al., 2008]	Container, Group, Chain

Table 3.8 – Vocabulary aligned with the type *Collection*.

Trace: We recovered exemplary vocabulary for the type *Trace* in Table 3.9, such as the term 'Weaving Model' that links to model elements from two different models. In summary, we consider vocabulary that relates to collection of pairs of references. We identify more general vocabulary such as 'identifier' and 'Reference'. In order to relate to the general notion of traceability, we align identified vocabulary with the type *Trace*. A trace is not explicitly considered in [Favre et al., 2012c]. In Table 3.9, we included papers that use vocabulary related to referencing, weaving models as well as trace links.

Paper	Keywords
[Simmonds et al., 2015]	Weaving model, annotation model
[Seibel et al., 2010]	TraceabilityNode, TraceabilityElement, TraceabilityLink, FactLink, ObligationLink, TraceabilityLinkType
[Jouault et al., 2010]	Trace Relationship, ModelWeaving, Relationship, ModelTraceRelationship
[Kling et al., 2011]	WeavingModel
[Méndez-Acuña et al., 2013]	WeavingModel, path, uri, Reference, references, NamedElement, name
[Barbero et al., 2008]	locator, identifier
[Toure et al., 2017]	Reference

Table 3.9 – Vocabulary aligned with the type *Trace*.

Concept: In Table 3.10, we group terms occurring in megamodels that represent concepts. They reflect on a structural or behavioral property, for example, views and viewpoints as known from software architecture modeling [Favre, 2004] or design patterns [Lämmel et al., 2014b]. Such concepts are typically documented, e.g., in a software documentation or Wikipedia.

Paper	Keywords
[Favre, 2004]	Meta-usecase, Viewpoint, View
[Lämmel et al., 2014b]	Concept, Feature

Table 3.10 – Vocabulary aligned with the type *Concept*.

Others: In many papers we find more non-frequent, domain-specific types of entities, for example, representing chrestomathy details, persons, monitors, execution details, control flow structures or repositories. As the most intriguing detail, the term 'Object' in [Seibel et al., 2010] does not relate to the context of an object graph at runtime but an object in the real world. We also identify terms for resources with an informative purpose, for example, web resources that resolve to documentation documents and additional informative data sets, i.e., metadata. We collect such other vocabulary in Table 3.11.

Paper	**Keywords**
[Favre, 2004]	Concern
[Barbero et al., 2008]	Metadata
[Vogel and Giese, 2012]	Additional Notes in the metamodel, Control Flow: MegamodelCall, InitialOperation, DecisionOperation, FinalOperation, ControlOperation, OperationTransition, Condition; Runtime Details: RuntimeEnvironment, ExecutionInformation and ExecutionContext
[Simmonds et al., 2015]	Stakeholder as Person
[Lämmel et al., 2014b]	Contribution, Contributor, Theme, Vocabulary
[Seibel et al., 2010]	Object, Characteristic
[Méndez-Acuña et al., 2013]	Attribute, MultiplicityElement
[Toure et al., 2017]	Dimension: data, control, presentation, platform, process; Objective: prescriptive, descriptive; Level: Terminal, Reference

Table 3.11 – Other vocabulary.

Types of Relations

Next, we offer the vocabulary for relations from the surveyed papers as summarized by Table 3.12. Terms related to equality and part-hood are ignored, because they would rather belong to an upper ontology such as DOLCE [Gangemi et al., 2002]. Such vocabulary is general and not specific to any domain.

Paper	**Relation Types**										
	Conformance	*Definition*	*Correspondence*	*Implementation*	*Usage*	*Membership*	*Typing*	*Function App.*	*Dependency*	*Abstract rel.*	*Other*
[Favre et al., 2012c]	x	x	x	x		x	x	x	x	x	
[Favre, 2004]	x	x	x	x	x				x		x
[Favre and Martinez, 2006]								x			x
[Gasevic et al., 2007]		x									x
[Barbero et al., 2008]										x	x
[Seibel et al., 2010]				x			x				x
[Jouault et al., 2010]										x	
[Kling et al., 2011]	x	x					x	x		x	
[Vogel and Giese, 2012]					x			x			x
[Méndez-Acuña et al., 2013]							x				
[Vignaga et al., 2013]	x									x	
[Lämmel et al., 2014b]				x	x		x		x		x
[Simmonds et al., 2015]	x							x			
[Salay et al., 2015]						x	x	x		x	
[Lämmel, 2016]			x			x					x
[Toure et al., 2017]	x					x					x
[Stevens, 2018]	x										
[Sottet et al., 2018]	x										x

Table 3.12 – Relation types in relevant papers discovered by the survey. The first line refers to the paper used for determining an initial set of vocabulary for common relations. The last four lines contain the update, i.e., four papers that are not used in [Heinz et al., 2017].

Conformance: We identify the term 'conformsTo' for a *conformance* relationship between an instance and its schema in all except one identified paper. Only in [Stevens, 2018], several subtypes of the relation are introduced.

Paper	Keywords
[Favre et al., 2012c]	conformsTo
[Simmonds et al., 2015]	conformsTo
[Favre, 2004]	conformsTo
[Kling et al., 2011]	conformsTo
[Vignaga et al., 2013]	conformsTo
[Sottet et al., 2018]	conformsTo
[Stevens, 2018]	conforms, safe-conforms, roundtrip-conforms

Table 3.13 – Vocabulary related to a conformance relation.

Definition: We find several megamodels, where an entity is linked to another because it provides an either formal or informal *definition*. For example in [Kling et al., 2011], a transformation model is linked to a transformation entity. A transformation model defines a transformation in the same way as a program defines a function or a grammar defines a language [Favre et al., 2012c]. We give an overview of related terms for the relation in Table 3.14.

Paper	Keywords
[Favre et al., 2012c]	definitionOf
[Favre, 2004]	isBasedOn, describedBy
[Kling et al., 2011]	association between Transformation and TransformationModel
[Gasevic et al., 2007]	extension (extensional definition of a system)

Table 3.14 – Vocabulary related to a definition relation.

Correspondence: We identify two different terms that express that two artifacts are largely equivalent. This makes it a specific kind of consistency relation that interconnects two artifacts that correspond to each other [Lämmel, 2016]. As depicted in Table 3.15, we find the explicit term 'correspondsTo' in two papers related to this thesis.

Paper	Keywords
[Favre et al., 2012c]	correspondsTo
[Favre, 2004]	isUsedtoCover
[Lämmel, 2016]	correspondsTo

Table 3.15 – Vocabulary related to a correspondence relation.

Implementation: Some papers include a megamodel of scenarios in which an artifact is considered to 'implement' or 'realize' another entity. For example, a parser may implement a language [Favre et al., 2012c] or a model element realizes a policy [Seibel et al., 2010]. Summarizing, a conceptual entity is implemented by a digital entity. We align such vocabulary as presented in Table 3.16.

Paper	Keywords
[Favre, 2004]	realizationOf
[Favre et al., 2012c]	implementedBy, describedBy
[Lämmel et al., 2014b]	implements
[Seibel et al., 2010]	represents

Table 3.16 – Vocabulary related to an implementation relation.

Usage: In literature, megamodels describe entities that either rely on the functionality of another entity or some kind of principles represented by another entity. In this context we specifically inspect the usage of concepts such as Ruby on Rails to support REST [Lämmel et al., 2014b]. We align the terms 'supports' and 'uses' as summarized in Table 3.17.

Paper	Keywords
[Vogel and Giese, 2012]	ModelUse
[Favre, 2004]	handledBy
[Lämmel et al., 2014b]	uses, supports

Table 3.17 – Vocabulary related to a usage relation.

Typing: We identify relations that express typing information as in the domain or range of a function. For example, in [Lämmel et al., 2014b] and [Kling et al., 2011] instantiation and subtyping are covered. We find typing for transformations in [Salay et al., 2015] as well as mathematical functions in [Favre et al., 2012c, Méndez-Acuña et al., 2013]. Such vocabulary as depicted in Table 3.18 is aligned with typing.

Paper	Keywords
[Favre et al., 2012c]	domainOf, hasRange
[Lämmel et al., 2014b]	instanceOf, isA
[Seibel et al., 2010]	isOfType
[Salay et al., 2015]	TypedNode's attribute type
[Kling et al., 2011]	extends (possibly enables subtyping), targetReferenceModel, srcReferenceModel
[Méndez-Acuña et al., 2013]	source,target, ArtifactType
[Toure et al., 2017]	type in Model

Table 3.18 – Vocabulary for relations related to typing information.

Function Application: In literature, the application of a function, such as a transformation, which relates to input and output artifacts can be modeled explicitly. Either, the application is captured as an extra node or it is encoded using 'input'/'output' relations as summarized for the surveyed papers in Table 3.19.

Paper	Keywords
[Favre et al., 2012c]	FunctionApplication, inputOf, hasOutput
[Vogel and Giese, 2012]	ModelUse, Input,Output
[Simmonds et al., 2015]	Transformation Record
[Salay et al., 2015]	TransformationApplication, TransformationMegaApp
[Favre and Martinez, 2006]	Refinement PIM-PSM, Refinement PSM-ISM
[Kling et al., 2011]	TransformationRecord, srcModel, targetModel
[Toure et al., 2017]	Input Model, Output Model

Table 3.19 – Vocabulary related to function application.

Membership: The membership relation has been mainly introduced in the context of linguistic architecture [Favre et al., 2012c], where an artifact is an element of a set. For example, a '.java'

file is an element of the set of all Java files. We only identify membership in other contexts as for example for function application, where functions are explicitly typed. If a function is typed by domain and range, it can only be applied to proper elements of the domain and create outputs that are elements of the range. In Table 3.20, we hence include papers that use vocabulary for typed functions and function application.

Paper	Keywords
[Favre et al., 2012c]	subsetOf, elementOf
[Salay et al., 2015]	in, out
[Kling et al., 2011]	sourceOf, targetOf, source, target
[Toure et al., 2017]	Input Model, Output Model

Table 3.20 – Vocabulary related to membership.

Dependency: The term *dependsOn* is considered for a relation in several papers [Favre et al., 2012c, Favre, 2004, Lämmel et al., 2014b]. We group the exact terms as depicted in Table 3.21 as dependency relations.

Paper	Keywords
[Favre et al., 2012c]	dependsOn
[Favre, 2004]	dependsOn
[Lämmel et al., 2014b]	dependsOn

Table 3.21 – Vocabulary related to a dependency relation.

Unspecified Relations: We also find several megamodels, where the visual representation depicts that there exists a relationship between two entities, but it is annotated with an arbitrary name like 'Relationship'. For example, a class called *Relationship* is given in [Kling et al., 2011], which is then associated with the class *Identified Element*. In papers such as [Kling et al., 2011] the relation is not specified in any concrete manner. Such appearances are listed in Table 3.22.

Paper	Keywords
[Favre et al., 2012c]	Relation
[Salay et al., 2015]	Relationship, Megarel
[Jouault et al., 2010]	Relationship, DirectedRelationship, source, sourceOf, target, targetOf, relatedTo, linked
[Barbero et al., 2008]	Relationship
[Kling et al., 2011]	Relationship, DirectedRelationship, linked, relatedTo
[Vignaga et al., 2013]	Relationship
[Sottet et al., 2018]	relatedTo

Table 3.22 – Vocabulary for unspecified relations.

Others: In analogy to entity types, there are many relation names, for which no common group can be identified. We present the terms in Table 3.23. In contrast to the terms for unspecified relations in Table 3.22, the relations are specified with a concrete semantics. Examples range from control flow issues [Vogel and Giese, 2012], over attributes of metadata [Barbero et al., 2008] to forms of abstraction [Gasevic et al., 2007] and model diffs [Lämmel, 2016].

Paper	**Keywords**
[Favre, 2004]	executes, has
[Barbero et al., 2008]	metadata attribute in Element
[Vogel and Giese, 2012]	mapsTo concerned with control-flow and further structural relations.
[Lämmel et al., 2014b]	documentedBy, developedBy, varies, moreComplexThan, similarTo
[Seibel et al., 2010]	in and out concerned with linking, elements and subElement and subLink as part-hood relation
[Favre and Martinez, 2006]	Structural relations between models and activities
[Gasevic et al., 2007]	RepresentationOf as Abstraction, Set(ExtensionalSystem), IntensionalSystem, intension (intensional definition of a system)
[Lämmel, 2016]	consistency, delta

Table 3.23 – Vocabulary for *Other* relations.

3.2 Static Semantics

When developers want to learn how to use a software technology, understanding the underlying conceptual ideas is crucial. In order to investigate the vocabulary used for expressing such conceptual knowledge in megamodels, we study common types of entities and relations. This section proceeds by discussing laws in terms of formal axioms for common vocabulary in the context of language-centric megamodels. Thus, we describe and formalize idioms that exist in software documentation from the perspective of a linguistic architecture of a software [Favre et al., 2012c]. Thereby, we answer the following research question:

> *RQ A2: What modeling idioms exist for (language-centric) megamodels that can be formalized as axioms?*

In the following subsections, we discuss core types of entities and relations. We formalize common laws for artifacts and their relatedness to conceptual entities, such as languages. Each law is illustrated by examples drawn from EMF usage [Steinberg et al., 2008]. *While the axioms are largely published in [Heinz et al., 2017], we raise awareness of minor differences to reviewed contributions in footnotes.*

To demonstrate how to use the vocabulary in models of an application, we state exemplary facts on the demo application called *SimplePO*, which can be found online.[4] This way, we intend to improve an overall understanding of the vocabulary for documenting software through megamodels with a focus on software languages and software technologies. As a methodological concern for developing this core ontology, competencies of types and relations are covered in terms of explicit *competency questions* [Sure et al., 2002, Scherp et al., 2011]. The competency questions are inspired by those presented in [Härtel et al., 2017], e.g., 'What is the language of each artifact?'. Since more entity types and relations are discussed here and the focus is on a rigorous formal axiomatization, we extend the set of competency questions from [Härtel et al., 2017].

We have implemented the axioms in Prolog including the examples as knowledge bases and provide the corresponding code snippets. The implementation and tests can be found on GitHub[5]. This Prolog project also includes a small model for the technology Django[6] by which we test theoretical aspects on inferring facts, which are discussed in Subsection 3.2.14.

[4] `https://www.informit.com/content/images/9780321331885/downloads/examples.zip` — Requested March 31, 2022

[5] `https://github.com/softlang/megaaxioms` — Requested March 31, 2022

[6] `https://www.djangoproject.com/` — Requested March 31, 2022

3.2.1 Artifacts

Every software project consists of a set of artifacts. To better distinguish artifacts in general from specific models, e.g., UML models, we only use the term *model* when we actually speak of a model in terms of a purpose-based subtype of artifact. Thus, we focus on *Artifact* as the basic type.

We distinguish several subtypes of *Artifact*, where files and folders are represented as instances of the types *File* and *Folder*[7]. We also introduce the subtype *Transient* for artifacts that only exist during program execution. As another addition, we introduce the subtype *Fragment* for artifacts that only exist as parts of files and transients.

The formal axioms, the Prolog implementation and documented instances related to EMF usage are given in Table 3.24. In summary, the following competencies are covered:

- *What artifacts exist in the software?*
- *How do artifacts manifest?*

Axioms	**Prolog**
$Artifact(e) \Rightarrow Entity(e).$ $Folder(a) \Rightarrow Artifact(a).$ $File(a) \Rightarrow Artifact(a).$ $Fragment(a) \Rightarrow Artifact(a).$ $Transient(a) \Rightarrow Artifact(a).$	`entity(X) :- artifact(X).` `artifact(X) :- folder(X).` `artifact(X) :- file(X).` `artifact(X) :- fragment(X).` `artifact(X) :- transient(X).`
EMF	
`folder("org.eclipse.emf.ecore").` *% metametamodel is a folder.* `folder("com.example.po").` *% the Java object model package is a folder.* `file("org.eclipse.emf.ecore.EObject.java").` *% the class EObject is a file.* `file("org.eclipse.emf.ecore.EPackage.java").` *% the class EPackage is a file.* `file("SimplePO.ecore").` *% the Ecore model for purchase orders is a file.* `file("SimplePO.genmodel").` *% the generator model is a file.* `transient(christmas_order_object).` *% the purchase order object is a transient.* `file("christmas_simplepo.xmi").` *% the persisted purchase order object is a file.* `fragment("SimplePO.ecore/PurchaseOrder").` *% the PurchaseOrder EClass is a fragment.*	

Table 3.24 – Formalization of types of digital artifacts.

[7] Compared to [Heinz et al., 2017], we drop *WebResource* and *Specification* to abstract from deployment location and purpose.

3.2.2 Systems and Technologies

We introduce *System* as a conceptual type for relating to a software as a whole. Any software system can be composed of (sub-) systems. In a description of a software, other software systems can be mentioned that are reused. Many times, reused software systems are not integral parts of the software. When a reused software systems is independent of a business use case and focuses on technical use cases for many business (or again technical) use cases, we speak of a technology. Hence, we also introduce the subtype *Technology*. A technology is again a system. Technologies, which includes frameworks and libraries, provide functionality that is meant to be reused by different systems—possibly several times in each system. A system is also composed of artifacts. While a system realizes a set of use cases, a subsystem or artifact only covers a subset of the use cases. One use case often only requires specific facets and thus dedicated parts of a system or a technology [Roover et al., 2013].

The axioms that introduce the types *System* and *Technology* are given in Table 3.25. As an illustration, we introduce several systems and technologies related to EMF. We provide examples for subsystems of the *SimplePO* software system that cover specific use cases as well as subsystems of the technology *EMF* that represent specific facet, such as persistence. The examples illustrate that there is no simple one-to-one matching from a subsystem in a software to a subsystem of a technology. We state the following competencies:

- *Which systems are reused by which other system?*
- *Which technologies are reused?*
- *Which subsystems need to be distinguished in order to differentiate between different business or technical use cases?*

Axioms	**Prolog**
$System(e) \Rightarrow Entity(e).$ $Technology(e) \Rightarrow System(e).$	`entity(X) :- system(X).` `system(X) :- technology(X).`
EMF	
`system(simplepo_app).` *% the SimplePO demo app is a system.* `system(simplepo_app_model).` *% SimplePO has a model subsystem.* `system(simplepo_app_edit).` *% SimplePO has an edit subsystem.* `technology(javac).` *% javac is a technology.* `technology(jvm).` *% JVM is a technology.* `technology(emf).` *% EMF is a technology.* `technology(emf_core).` *% EMF has a core subsystem.* `technology(emf_persistence).` *% EMF has a persistence subsystem.* `technology(emf_code_generator).` *% EMF has a code generator subsystem.*	

Table 3.25 – Software is represented as a conceptual entity that instantiates the type *System*. Its specialization *Technology* represents a system that is reusable in multiple occasions.

3.2.3 Part-hood Integrity

A basic ontological definition of part-hood can be found in DOLCE [Gangemi et al., 2002]. Integrating part-hood relations in an ontology is non-trivial [Guizzardi, 2009]. Part-hood may not be transitive when breaching multiple domain classes. For example, Rio de Janeiro may be *part of* Brazil and Brazil may be *part of* the organization UN, but Rio de Janeiro is typically not seen as a *part of* the UN. To address this problem, we carefully refine part-hood and use different relations for different kinds of part-hood. We discuss and elaborate on this problem in the conclusion in Section 3.3.

The formalization of part-hood can be reused from DOLCE, but we add domain-specific integrity constraints inspired by [Tran and Debruyne, 2012] as presented in Table 3.26. They refine part-hood between different domain classes so that it is used as a transitive relation. There exists an order of magnitude for types of artifacts. Instances of a larger magnitude cannot be parts of instances of a smaller magnitude. Thus, only fragments can be parts of fragments, a system cannot be part of an artifact, but a system can have all types of artifacts and systems as parts. Such integrity of part-hood for artifacts and systems is formalized in Table 3.26. Furthermore, a fragment cannot exist on its own but needs another artifact that it is part of. The examples for *SimplePO* with respect to the use of *EMF* illustrate part-hood along the different orders of magnitude.

Below, we state competency questions related to part-hood. The general competencies are provided for DOLCE [Gangemi et al., 2002]. We elaborate on more focused competencies that are only expressed in a generalized fashion elsewhere:

- *What parts exist in a system?*
- *In which artifact can we find a fragment?*
- *What is the part-hood hierarchy for systems and artifacts?*

Axioms	Prolog
$Fragment(p)$ $\Rightarrow \exists w.Artifact(w) \wedge partOf(p,w).$ $partOf(p,w) \wedge System(w)$ $\Rightarrow (System(p) \vee Artifact(p)).$ $partOf(p,w)$ $\wedge Artifact(w) \wedge \neg Fragment(w)$ $\Rightarrow Artifact(p).$ $partOf(p,w) \wedge Fragment(w)$ $\Rightarrow Fragment(p).$	`ok_type(fragment(F)):-` `  part_of(F,A), artifact(A).` `ok_relation(part_of(P,W)):-` `  system(W), (artifact(P); system(P)),` `  not(part_of(W,P);W==P).` `ok_relation(part_of(P,W)):-` `  artifact(W), not(fragment(W)),` `  artifact(P).` `ok_relation(part_of(P,W)):-` `  fragment(W), fragment(P).` `part_ofT(A1,A1).` `part_ofT(A1,A2) :- part_of(A1,A2).` `part_ofT(A1,A3) :-` `  part_of(A1,A2),` `  part_ofT(A2,A3).`
EMF	
`part_of(simplepo_app_edit,simplepo_app).` `part_of(simplepo_app_model,simplepo_app).` `part_of("SimplePO.ecore",simplepo_app_model).` `part_of("SimplePO.ecore/PurchaseOrder","SimplePO.ecore").` `part_of("SimplePO.genmodel", simplepo_app_model).` `part_of("christmas_simplepo.xmi", simplepo_app_model).` `part_of(emf_core,emf).` `part_of(emf_persistence,emf).` `part_of("org.eclipse.emf.ecore",emf_core).` `part_of("org.eclipse.emf.ecore.EObject.java","org.eclipse.emf.ecore").` `part_of("org.eclipse.emf.ecore.EPackage.java","org.eclipse.emf.ecore").`	

Table 3.26 – Aside from the reused semantics for *partOf* as provided in DOLCE [Gangemi et al., 2002], we include entity type-specific constraints. We consider *partOf* to be a reflexive, antisymmetric, transitive relation. In the Prolog implementation, we use *partOfT* as the transitive and reflexive variant of *partOf*. We use the suffix T as a convention for encoding any reflexive and transitive relation.

3.2.4 Languages

For software languages, we introduce the type *Language*. When we speak of software languages, we mean syntactic structures for which syntax, type system, semantics and pragmatics can be (in-)formally defined [Favre et al., 2009, Lämmel, 2018]. In linguistic architecture, software languages are used to classify artifacts. An artifact's membership to a language is expressed by the relation *elementOf*. Every artifact is an element of a language.[8]

In the formalization, we rely on set theory. The formal axioms for the type *Language* and the relation *elementOf* are given in Table 3.27. The examples introduce several common languages such as *Java*. A software language is a set of artifacts. Comments are given for the examples to help with understanding the main idea with using set theory for formalization. We state the following competency questions that are addressed by the type *Language* and the relation *elementOf*.

- *What are the different languages that can be identified in a software system?*
- *What artifacts are elements of a software language?*

Axioms	**Prolog**
$Language(l) \Rightarrow Entity(l).$ $elementOf(a, l)$ $\Rightarrow Artifact(a) \wedge Language(l).$ $Artifact(a) \Rightarrow \exists l.elementOf(a, l).$	`entity(X) :- language(X).` `ok_relation(element_of(A,L)):-` `  artifact(A), language(L),` ... `ok_type(artifact(A)):-` `  element_of(A,_).`
EMF	
`language(xmi).` *% the set of all XMI artifacts* `language(java).` *% the set of all Java artifacts* `language(java_package).` *% the set of all Java packages* `language(jvm_objects).` *% the set of all objects at JVM runtime* `element_of("org.eclipse.emf.ecore",java_package).` `element_of("org.eclipse.emf.ecore.EObject.java",java).` `element_of("org.eclipse.emf.ecore.EPackage.java",java).` `element_of("SimplePO.ecore",xmi).` `element_of("SimplePO.genmodel",xmi).` `element_of(christmas_order_object,jvm_objects).` `element_of("christmas_simplepo.xmi",xmi).` `element_of("com.example.po",java_package).`	

Table 3.27 – The table presents the axioms for software languages as an entity type as well as the membership relation *elementOf*. The membership relation is refined later.

[8] Compared to [Heinz et al., 2017], we drop *elementOf* for pairs of artifacts related to function application and later introduce new dedicated vocabulary.

3.2.5 Subsets of Languages

A software language can be subdivided into subsets. Here, we are most interested in the competency of a language to classify an artifact in the linguistic dimension in a more precise manner [Favre et al., 2012c].

The asymmetric relation *subsetOf* is the basic relation for a subset being included in a set, here, a software language. *subsetEqOf* extends the relation by reflexivity and transitivity.[9] In Table 3.28, we provide examples for technology-specific subsets and their members. Again comments help with understanding the set theoretical idea behind the axioms. The following competency question defines the purpose of the subset relation.

- *Is an artifact a member of a more specific subset of a software language?*

Axioms	**Prolog**
$subsetOf(l, l')$ $\Rightarrow Language(l) \wedge Language(l').$ $subsetOf(l, l')$ $\Rightarrow \neg subsetOf(l', l).$ $subsetEqOf(l, l') \wedge subsetEqOf(l', l'')$ $\Rightarrow subsetEqOf(l, l'').$ $subsetEqOf(l, l')$ $\Rightarrow Language(l) \wedge Language(l').$ $subsetEqOf(l, l')$ $\Rightarrow l = l' \vee subsetOf(l, l')$ $\vee \exists ll.subsetOf(l, ll) \wedge subsetEqOf(ll, l').$	`ok_relation(subset_of(L1,L2)):-` `  language(L1), language(L2),` `  not(subset_of(L2,L1)).` `ok_relation(subset_ofT(L1,L1)):-` `  language(L1).` `ok_relation(subset_ofT(L1,L2)):-` `  subset_of(L1,L2).` `ok_relation(subset_ofT(L1,L2)):-` `  subset_of(L1,L), subset_ofT(L,L2).`
EMF	
`language(ecore_java).` *% the set of all Java artifacts derived from an Ecore model* `language(ecore_xmi).` *% the set of all Ecore models* `language(genmodel_xmi).` *% the set of all generator models* `language(simplepo_xmi).` *% the set of all purchase orders in XMI* `language(ecore_java_package).` *% the set of all java packages based on an ecore model* *% Subset relationships* `subset_of(ecore_xmi,xmi).` `subset_of(simplepo_xmi,xmi).` `subset_of(ecore_java,java).` `subset_of(ecore_java_package,java_package).` *% Subset membership* `element_of("SimplePO.ecore",ecore_xmi).` `element_of("SimplePO.genmodel",genmodel_xmi).` `element_of("christmas_simplepo.xmi",simplepo_xmi).` `element_of("com.example.po",ecore_java_package).`	

Table 3.28 – The table presents the axioms for the subset relation and provides examples for subset membership relationships. In the Prolog implementation, we use a distinct transitive variant of subset relation (*subset_ofT*) and a non-transitive variant (*subset_of*).

[9] Compared to [Heinz et al., 2017], we clarify transitivity for *subsetEqOf* and add more dedicated examples for members of subsets.

3.2.6 Functions

Artifacts may depend on other artifacts in various ways. Functions can produce output artifacts based on input artifacts. For example, when using EMF, Java code is produced based on input models. We define the type *Function* as a mapping between sets of input artifacts and sets of output artifacts.

We again provide axioms that rely on set-theory. Conceptually speaking, a function connects input artifacts that are members of a domain with output artifacts that are members of a range. The domain and range are hence (tuples/sequences of) sets of artifacts, i.e., (tuples/sequences of) languages. The relation *funType* connects a function with its domain and range, which represent its type. Every function has a type.[10] We provide a conceptual illustration in Figure 3.1, where an exemplary type of a function is included above the lower border of the domain and range bubble. It relates to code generation for an EMF use case, in which Ecore models written in the language *EcoreXMI* and generator models written in *GeneratorXMI* are mapped to an implemented subset of Java that we name *EcoreJava*. The axioms and further examples are provided in Table 3.29.

We provide the competency questions on functions below:

- *What functions express a dependency between sets of input and output artifacts?*
- *What languages are the in- and output sets of a function and thus form its type?*

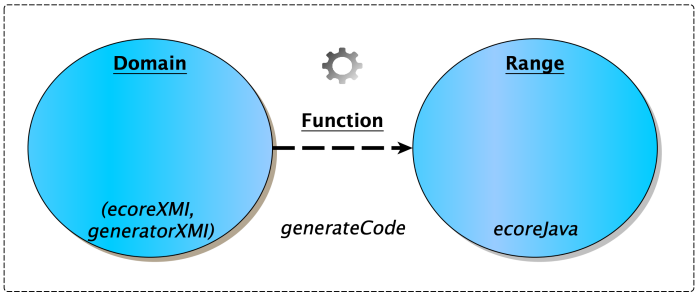

Figure 3.1 – Conceptual illustration on the formalization of functions. Thereby, we emphasize the analogy to a mathematical function that connects a typically numeric domain and range.

[10] In comparison to [Heinz et al., 2017], we repair the disconnect between domain and range of a function by stating a function's type. Furthermore, instead of using a predicate to encode tuples/sequences, we use tuples directly as syntactic parts of the notation.

Axioms	**Prolog**
$Function(f) \Rightarrow Entity(f).$ $Function(f)$ $\Rightarrow \exists ds, rs.funType(f,(ds,rs)).$ $funType(f,(ds,rs))$ $\Rightarrow Function(f)$ $\wedge \forall d \in ds : Language(d)$ $\wedge \forall r \in rs : Language(r).$	*entity*(X) :- *function*(X). ok_type(*function*(F)):- *fun_type*(F,_), ... ok_relation(*fun_type*(F,(Ds,Rs))):- *function*(F), forall(member(D,Ds), *language*(D)), forall(member(R,Rs), *language*(R)).
EMF	
%function declaration *function*(save_model). *fun_type*(save_model,([jvm_objects],[xmi])). *function*(load_model). *fun_type*(load_model,([xmi],[jvm_objects])). *function*(generate_code). *fun_type*(generate_code,([ecore_xmi,genmodel_xmi],[ecore_java_package])). *fun_type*(generate_code,([eclass_xmi,genclass_xmi], [ecore_java])).	

Table 3.29 – We formalize functions and their type where the type is a pair of domain- and range tuples. The full Prolog code with regard to constraints for the type *Function* is refined later.

3.2.7 Function Application

In the previous subsection, we have covered how functions are typed. Here, we formalize how functions are applied. A function application is a mapping between artifacts that can be validated against the function's type. Input artifacts are produced based on output artifacts. At this point, we also clarify that artifacts do not only manifest as files, folders, and fragments. The type *Transient* proxies for artifacts that occur as (parts of) intermediate results—produced by functions.

While *funType* is a relation between the function and its domain and range, the relation (*funApply*) for function application connects the function entity with concrete elements of the domain and range.[11] EMF-based code generation is a commonly applied function. In summary, EMF-specific Java code is derived from an Ecore model and a generator model. It is provided as an example in Table 3.30. The examples also relate to persistence and introduce *turnstileJVMObject* as an exemplary transient. We consider the following competencies:

- *Which artifacts are derived from which other artifacts?*
- *Which artifacts are transients?*

[11] In comparison to [Heinz et al., 2017], we omit mentioning sequence and tuples completely while we still use them in the formalization itself. For example, the input artifacts of a function are grouped by a tuple.

Axioms	Prolog
$funApply(f,(is,os))$ $\Rightarrow Function(f)$ $\wedge\forall i.i \in is : Artifact(i)$ $\wedge\forall o.o \in os : Artifact(o).$ $funApply(f,(is,os))$ $\Rightarrow \exists ds,rs.imageOf((ds,rs),f)$ $\forall i.0 \leq i < \lvert ds\rvert.elementOf(is_i,ds_i)$ $\forall j.0 \leq j < \lvert rs\rvert.elementOf(os_j,rs_j).$ $Transient(t)$ $\Rightarrow \exists f,os,o.$ $\wedge funApply(f,(is,os))$ $\wedge o \in os$ $\wedge(t == o \vee partOfT(t,o)).$	`ok_relation(fun_apply(F,(IS,OS))):-` `  function(F),` `  forall(member(I,IS), artifact(I)),` `  forall(member(O,OS), artifact(O)),` `  fun_type(F,(DS,RS)),` `  zip(IS,DS,IDs),` `  forall(member((I,D),IDs),` `  element_ofT(I,D)),` `  zip(OS,RS,ORs),` `  forall(member((O,R),ORs),` `  element_ofT(O,R)).` `ok_type(transient(T)):-` `  fun_apply(_,(_,OS)),` `  member(T1,OS),` `  part_ofT(T,T1).`
EMF	

```
%function application
fun_apply(save_model,
    ([christmas_order_object],["christmas_simplepo.xmi"])).
fun_apply(load_model,
    (["christmas_simplepo.xmi"],[christmas_order_object])).
fun_apply(generate_code,
    (["SimplePO.ecore", "SimplePO.genmodel"],["com.example.po"])).
% function application on fragment level
fun_apply(save_model,
  (["christmas_order_object.purchaseOrder"],
  ["christmas_simplepo.xmi/PurchaseOrder"])).
fun_apply(save_model,
  (["christmas_order_object.purchaseOrder.item[0]"],
  ["christmas_simplepo.xmi/PurchaseOrder/item[0]"])).
fun_apply(save_model,
  (["christmas_order_object.purchaseOrder.item[1]"],
  ["christmas_simplepo.xmi/PurchaseOrder/item[1]"])).
fun_apply(generate_code,
  (["SimplePO.ecore/Item","SimplePO.genmodel/Item"],
  ["Item.java"])).
fun_apply(generate_code,
  (["SimplePO.ecore/PurchaseOrder","SimplePO.genmodel/PurchaseOrder"]
  ,["PurchaseOrder.java"])).
```

Table 3.30 – We formalize function application. A function application relates to the conceptual function entity and an input/output pair. In the Prolog implementation, we use the zip predicate to match inputs with domains and outputs with ranges based on their order of appearance. Then, we check for every pair that the input/output is a member of the domain/range or a transitive member, because it is member of a subset of the domain/range (See Subsection 3.2.14).

3.2.8 Definition and Implementation

Function and *Language* are conceptual types. For each instance, there is a formal or at least semi-formal definition or an implementation. There may exist both at the same time; a specification artifact that defines it and code as part of a system or artifact that implements it. Hence, we may always aim to identify implementing and defining artifacts, when the documenting megamodel is meant to be used to explore the system that it documents.[12]

In Table 3.31, the axioms for *Language* and *Function* are completed. The examples address implementation of introduced functions and languages. The example on language definition is further documented to emphasize the following law. When a schema defines a language, we can use it to determine valid instances or, linguistically speaking, the members of the language. We state these competencies regarding the relations *defines* and *implements*:

- *Which system implements which language?*
- *Which system implements which function?*
- *Which artifacts implement/define which language?*
- *Which artifacts implement/define which function?*

[12]Compared to [Heinz et al., 2017], an artifact may now implement as well. This way, a system can always be replaced by more concrete artifacts that are part of it.

Axioms	Prolog
$defines(a, e) \Rightarrow Artifact(a) \wedge Entity(e).$ $implements(x, y)$ $\Rightarrow (Artifact(x) \vee System(x))$ $\wedge (Function(y) \vee Language(y)).$ $Language(l)$ $\Rightarrow \exists s.defines(s, l) \vee implements(s, l).$ $Function(f)$ $\Rightarrow \exists s.defines(s, f) \vee implements(s, f).$	`ok_relation(defines(A,E)):-` `  artifact(A), entity(E).` `ok_relation(implement(X,Y)):-` `  (artifact(X);system(X)),` `  (function(Y);language(Y)).` `ok_type(language(L)):-` `  defines(_,L); implement(_,L).` `ok_type(function(F)):-` `  fun_type(F,_),` `  (defines(_,F);implement(_,F)).`
EMF	
%implementation of functions `implement(emf_persistence,save_model).` `implement(emf_persistence,load_model).` `implement(emf_code_generator,generate_code).` *%implementation of languages* `implement(emf_persistence,ecore_xmi).` `implement(emf_persistence,genmodel_xmi).` `implement(emf_code_generator,ecore_java).` `implement(javac,java).` `implement(javac,java_package).` `implement(jvm,jvm_objects).` *% Definition of a subset* `language(simplepo_xmi).` *%element_of("christmas_simplepo.xmi",simplepo_xmi). is stated earlier*	

Table 3.31 – We formalize definition and implementation, where any language or function needs to be either defined or implemented. In the Prolog implementation we use 'implement' instead of 'implements' to avoid collision with the built-in predicate 'implements/2'.

3.2.9 Conformance

As explained in the previous subsection, an artifact may offer a formal definition of a language. We introduce *conformsTo* to cover conformance of an artifact to a definition such as a schema. In the broader sense, we also relate to types, as defined by type systems, as languages, because they define the set of typically transient values.

We provide an idealized static semantics. If the schema and the instance are composite, for every part of the instance exists a schema part such that the former conforms to the latter. Otherwise, the instance is not composite and it is an element of the language defined by the schema part. Actual conformance deviates from the idealized axiomatization, if the mapping from schema composites to instance components or vice versa is not deterministic or otherwise unclear. The examples in Table 3.32 walk through a traceable conformance scenario, where conformance is stated for the schema for simple purchase orders *SimplePO.ecore* and an instance with a purchase order for Christmas. We always introduce the parts of the schema and instance first and then add the *defines* and *elementOf* relationships to illustrate validity of the *conformsTo* relationships. We state the following competencies for the relation *conformsTo*.

- *Which schema artifact can be used to validate an instance artifact?*
- *Which parts of an artifact conform to which parts of a schema?*

Axioms	**Prolog**
$conformsTo(a, a')$ $\Rightarrow Artifact(a) \wedge Artifact(a')$ $conformsTo(a, a')$ $\Rightarrow (\exists l.defines(a', l) \wedge elementOf(a, l))$ $\vee(\forall p.partOf(p, a)\exists p'.partOf(p', a')$ $\wedge conformsTo(p, p'))$.	`ok_relation(conforms_to(A1,A2)):-` `  artifact(A1),artifact(A2),((` `  defines(A2,L),element_ofT(A1,L));` `  forall(part_of(P1,A1),(` `  part_of(P2,A2),` `  conforms_to(P1,P2)))).`
EMF	
%Traceable Conformance `conforms_to("christmas_simplepo.xmi","SimplePO.ecore").` `part_of("christmas_simplepo.xmi/PurchaseOrder","christmas_simplepo.xmi").` *%part_of("SimplePO.ecore/PurchaseOrder","SimplePO.ecore"). is stated earlier.* `conforms_to("christmas_simplepo.xmi/PurchaseOrder","SimplePO.ecore/PurchaseOrder").` `element_of("christmas_simplepo.xmi/PurchaseOrder",purchase_order_xmi).` `defines("SimplePO.ecore/PurchaseOrder",purchase_order_xmi).` `part_of("christmas_simplepo.xmi/PurchaseOrder/item[0]",` `  "christmas_simplepo.xmi/PurchaseOrder").` `part_of("christmas_simplepo.xmi/PurchaseOrder/item[1]",` `  "christmas_simplepo.xmi/PurchaseOrder").` `part_of("SimplePO.ecore/Item","SimplePO.ecore").` `conforms_to("christmas_simplepo.xmi/PurchaseOrder/item[0]","SimplePO.ecore/Item").` `conforms_to("christmas_simplepo.xmi/PurchaseOrder/item[1]","SimplePO.ecore/Item").` `defines("SimplePO.ecore/Item",purchase_order_item_xmi).` `element_of("christmas_simplepo.xmi/PurchaseOrder/item[0]",purchase_order_item_xmi).` `element_of("christmas_simplepo.xmi/PurchaseOrder/item[1]",purchase_order_item_xmi).`	

Table 3.32 – We formalize an idealized conformance. The formalization states that every part of a conforming artifact needs to conform to a part of the schema. In the Prolog implementation on GitHub, we first introduce entities such as *christmas_simplepo.xmi/order/item[0]* as valid fragments. Hence, the Prolog code includes a few more facts for the sake of completeness while the facts essential to understanding conformance are stated here.

3.2.10 Correspondence

Correspondence expresses that two artifacts that can be members of different languages represent largely the same data. The relation is often used in documentation of systems that either support or use data mapping, for example, *EMF* or *Hibernate*[13].

We axiomatize an idealized correspondence of artifacts x and y as follows. If the artifacts are composite, then for each part of x there is a corresponding part of y and vice versa. Otherwise, a value level is reached and both artifacts (parts) are equal. Such an idealized axiomatization does not consider issues due to even simple forms of an 'impedance mismatch' [Lämmel and Meijer, 2007]. For instance, an artifact can contain fragments that do not correspond to any part on the other side. The degree of nesting may also be different. An illustrative correspondence scenario is given in Table 3.33. It relates to the Christmas order serialized in a subset of *XMI* again, which we introduced earlier. Here, we make its correspondence to the runtime object traceable. For the relation *correspondsTo*, we state the following competencies.

- *Which artifacts correspond to each other?*
- *Which parts of two corresponding artifacts are the same or again correspond?*

Axioms

$$correspondsTo(a, a') \Rightarrow Artifact(a) \wedge Artifact(a').$$

$$\begin{aligned} &correspondsTo(a, a') \\ &\Rightarrow (\nexists p.partOf(p, a) \vee partOf(p, a')) \\ &\quad \wedge sameAs(a, a') \\ &\vee(\forall p.partOf(p, a) \exists p'.partOf(p', a') \\ &\quad \wedge correspondsTo(p, p')) \\ &\wedge(\forall p'.partOf(p', a') \exists p.partOf(p, a) \\ &\quad \wedge correspondsTo(p, p')) \end{aligned}$$

Prolog

```
ok_relation(corresponds_to(A1,A2)):-
  artifact(A1),artifact(A2),
  ok_directed(corresponds_to(A1,A2)),
  ok_directed(corresponds_to(A2,A1)).

ok_directed(corresponds_to(A1,A2)):-
  forall(part_of(P1,A1),(
    part_of(P2,A2),
    corresponds_to(P1,P2))),!.

ok_directed(corresponds_to(A1,A2)):-
  not(part_of(_,A1);part_of(_,A2)),
  same_as(A1,A2).
```

EMF

```
%Correspondence scenario
corresponds_to(christmas_order_object,"christmas_simplepo.xmi").
corresponds_to("christmas_order_object.purchaseOrder",
               "christmas_simplepo.xmi/PurchaseOrder").
corresponds_to("christmas_order_object.purchaseOrder.item[0]",
               "christmas_simplepo.xmi/PurchaseOrder/item[0]").
corresponds_to("christmas_order_object.purchaseOrder.item[1]",
               "christmas_simplepo.xmi/PurchaseOrder/item[1]").
same_as("christmas_order_object.purchaseOrder.item[0]",
        "christmas_simplepo.xmi/PurchaseOrder/item[0]").
same_as("christmas_order_object.purchaseOrder.item[1]",
        "christmas_simplepo.xmi/PurchaseOrder/item[1]").
```

Table 3.33 – We formalize correspondence while ignoring an impedance mismatch known in the technological space of (O/X-)mapping technologies. In the example, we present a correspondence trace in terms of illustrative matched resource parts between a Java object and a persisted XMI file. In the Prolog implementation, we use the helper predicate *ok_directed* to check directions of symmetric relations one by one.

[13] `https://hibernate.org/` — Requested March 31, 2022

3.2.11 Concepts in Software Engineering

We use the type *Concept* to address software engineering concepts, e.g., design patterns or protocols. Concepts are defined by artifacts (i.e., more or less formal specifications). This way, the formalization is similar to that of languages. Next, we introduce the relation *uses*. An entity uses a concept, if it complies to the concept's definition. Hence, we also introduce the relation *compliesTo*.[14] It is a more informal and less constrained variant of the relation *conformsTo*. In theory, we could derive a predicate from a concept's definition to decide on whether the usage conforms to the concept's specification. Since there may not exist a formal specification for every concept, the decision can be subjective and not as traceable as for conformance. Furthermore, we introduce the relation *facilitates* to express that a technology facilitates the usage of a concept by providing supportive means. EMF's persistence subsystem facilitates *XMI serialization* as an example for an operational concept. If a system uses a technology and the technology facilitates a concept, then the system uses this concept. Thus, facilitation implies a deferred usage.

The axioms are given in Table 3.34. The examples relate to *XMI serialization* as a concept, which is defined by an OMG standard. In the table, the example contains the link to the specification, which is modeled as an artifact in the knowledge base. Below, we state the competency questions with a focus on usage and facilitation of concepts.

- *What concepts are used by languages, technologies, systems, and artifacts?*
- *Which concepts are facilitated by which used technology?*
- *Which artifact defines a used concept?*
- *What definitions of concepts does an artifact comply to?*

[14] Compared to [Heinz et al., 2017], we introduce the relation *compliesTo* as a less formal variant of *conformsTo*, because the validation of laws for conformance is typically more difficult for informal natural language definition and may in the worst case require subjective interpretation.

Axioms	Prolog
$Concept(c) \Rightarrow Entity(c).$ $Concept(c) \Rightarrow \exists a.defines(a,c).$ $uses(s,t) \wedge Technoloy(t)$ $\Rightarrow \exists c.uses(s,c) \wedge facilitates(t,c).$ $uses(x,c) \wedge Concept(c)$ $\Rightarrow \exists s.defines(s,c) \wedge \exists p.partOfT(p,x)$ $\wedge compliesTo(p,s).$ $facilitates(t,c)$ $\Rightarrow Technology(t) \wedge Concept(c).$ $compliesTo(x,a)$ $\Rightarrow Artifact(a) \wedge Entity(x).$	*entity*(X) :- *concept*(X). ok_type(*concept*(C)):- *defines*(_,C). ok_relation(*uses*(S,T)):- *technology*(T), *facilitates*(T,C), (*uses*(S,C);*facilitates*(S,C)). ok_relation(*uses*(S,C)):- *concept*(C), *defines*(A,C), (part_ofT(P,S); *uses*(S,T),part_ofT(P,T)), *complies_to*(P,A). ok_relation(*facilitates*(T,C)):- *technology*(T), *concept*(C). ok_relation(*complies_to*(E,A)):- *artifact*(A), *entity*(E).
EMF	
concept(xmiserialization). *defines*("https://www.omg.org/spec/XMI/2.5.1/PDF",xmiserialization). *uses*(simplepo_app,xmiserialization). *uses*(simplepo_app,emf_persistence). *facilitates*(emf_persistence,xmiserialization). *complies_to*(emf_persistence,"https://www.omg.org/spec/XMI/2.5.1/PDF").	

Table 3.34 – We introduce integrity constraints on concepts, usage and facilitation. For the axioms on usage, we systematically follow along columns in Table 3.35.

3.2.12 Usage

Here, we reflect in more detail on the question on what types appear in a usage relation. A technology, system or artifact is used as soon as it is referred to. The using types are technology, system, and artifact. A technology, system or artifact can use a language. Then, at least a (transitive) part of the using side has to be written in this language. Concept usage is similar to language membership on a more abstract level. A system can use a concept that is defined by an artifact, e.g., a design pattern, if some parts of it conform to the concept. We specify the types that are usable in the domain or range of *uses* relationships in Table 3.35. In the competency questions below, we use the term 'depend' as a form of usage.

- *Which languages are used by a system or artifact?*
- *What are the technologies reused by systems, and, more specifically, by artifacts?*
- *What other systems do systems, and specific artifacts depend on?*
- *Which artifacts do systems, technologies and other artifacts depend on?*
- *What concept is used by which language, technology, system, or artifact?*

	Language	*Technology*	*System*	*Artifact*	*Concept*
Language					•
Technology	•	•	•	•	•
System	•	•	•	•	•
Artifact	•	•	•	•	•
Concept					•

Table 3.35 – Types usable in *uses* relationships, where the left side is the user.

Axioms	Prolog
$uses(x,y) \wedge System(y)$ $\Rightarrow Artifact(x) \vee System(x).$ $uses(x,y) \wedge Language(y)$ $\Rightarrow \exists p.partOfT(p,x)$ $\wedge elementOfT(p,y).$ $uses(x,y) \wedge Artifact(y)$ $\Rightarrow Artifact(x) \vee System(x).$	`ok_relation(uses(S1,S2)):-` `  system(S1), system(S2),` `  not(technology(S2)).` `ok_relation(uses(S,L)):-` `  language(L), part_ofT(P,S),` `  element_ofT(P,L).` `ok_relation(uses(S,A)):-` `  artifact(A),` `  (artifact(S); system(S)).`

EMF

```
% Language Usage with language members in line comments
uses(simplepo_app,simplepo_xmi).
%element_of("christmas_simplepo.xmi",simplepo_xmi). is stated earlier
uses(simplepo_app,ecore_java).
uses("Item.java",ecore_java).
%element_of("Item.java",ecore_java). is stated earlier
uses(emf,java).
%element_of("org.eclipse.emf.ecore.EObject.java",java). is stated earlier

%Technology usage
concept(code_generation).
uses(simplepo_app,code_generation).
technology(jet).
defines("https://en.wikipedia.org/wiki/Code_generation_(compiler)"
       ,code_generation).
uses(emf,jet).
uses(simplepo_app,emf).
facilitates(jet,code_generation).
facilitates(emf,code_generation).
complies_to(jet,"https://en.wikipedia.org/wiki/Code_generation_(compiler)").
complies_to(emf,"https://en.wikipedia.org/wiki/Code_generation_(compiler)").

%Artifact usage
uses(simplepo_app,"org.eclipse.emf.ecore.EObject.java").
uses("Item.java","org.eclipse.emf.ecore.EObject.java").
```

Table 3.36 – For the axioms on usage, we systematically follow along columns in Table 3.35. The axioms use the transitive variants of *partOf* and *elementOf*. Usage of systems is covered in examples since every technology is a system. In the examples, we use a Wikipedia page as an informal definition of code generation according to common knowledge.

3.2.13 Traceability

In a software, many artifacts contain references, where a reference often encodes a relationship. Thus, a reference can be used to navigate between artifacts that are in a relationship [Jouault et al., 2010]. For example, a Java class *uses* another class through subtyping. Then, it refers to the other class in its signature as in 'extends EObject'. Next, an *Ecore instance model*, e.g., *christmas_simplepo.xmi*, *conforms to* an *Ecore model*, e.g., *SimplePo.ecore*. It refers to the Ecore model that it conforms to. An EMF generator model refers to both, the related Ecore model and a folder, where the generated Java code is placed. The latter references provide evidence for a function application of the function *generate_code*.

We formalize *Reference* as well as *Trace* as types. A trace is a nested sequence of tuples of references[15]. Traces of relationships between artifacts can be identified, persisted, and used for navigation. Any reference is encoded by a fragment. Hence, we introduce the relation *encodedBy*. Any reference resolves to an artifact. Therefore, we also introduce the relation *resolvesTo*. The axioms are given in Table 3.37. The examples illustrate traceability for artifacts that are in a conformance relationship as well as artifacts that are related through function application of the function *generate_code*. Below, the competency questions summarize the assumed information need:

- *Which artifact encodes a reference?*
- *Which artifact does a reference resolve to?*
- *What trace can be used as evidence for a relationship between artifacts?*

[15] In comparison to [Heinz et al., 2017], we explicitly introduce the types *Reference* and *Trace*. Hence, the formalization of traceability is adapted.

Axioms	Prolog
$Reference(r) \Rightarrow Entity(r).$ $Reference(r) \Rightarrow \exists a : resolvesTo(r, a).$ $Sequence(s) \Rightarrow Entity(s)$ $Trace(s) \Rightarrow Sequence(s)$ $Trace(s) \Rightarrow \forall t \in s. \forall r \in t. Reference(r).$ $encodes(a, r)$ $\Rightarrow Artifact(a) \wedge Reference(r).$ $resolvesTo(r, a)$ $\Rightarrow Reference(r) \wedge Artifact(a).$ $resolvesTo(r, a)$ $\Rightarrow \nexists r'. resolvesTo(r', a) \wedge r' \neq r.$ $refersTo(a, a')$ $\Rightarrow \exists p. partOf(p, a) \wedge encodes(p, r)$ $resolvesTo(r, a').$	`entity(X) :- reference(X).` `entity(X) :- trace(X).` `ok_type(reference(R)):-` `  encodes(_,R), resolves_to(R,_).` `ok_type(trace(T)):-` `  forall(` `  (member(TUPLE,T),member(R,TUPLE)),` `  reference(R)).` `ok_relation(encodes(A,R)):-` `  artifact(A), reference(R).` `ok_relation(resolves_to(R,A)):-` `  reference(R), artifact(A).` `ok_relation(refers_to(A1,A2)):-` `  part_ofT(P,A1), encodes(P,R),` `  resolves_to(R,A2).`

EMF

```
% the instance model encodes a reference to the Ecore model
reference(simplepo_ref).
resolves_to(simplepo_ref,"SimplePO.ecore").
encodes("christmas_simplepo.xmi",simplepo_ref).

% the instance model encodes a reference to itself, because once we access
% the instance model, we know a qualified path to it.
reference(christmas_simplepo_ref).
resolves_to(christmas_simplepo_ref,"christmas_simplepo.xmi").
encodes("christmas_simplepo.xmi",christmas_simplepo_ref).

% Traceability Recovery conforms_to("christmas_simplepo.xmi","SimplePO.ecore").
trace([
 [christmas_simplepo_ref,simplepo_ref],
 [christmas_simplepo_order_ref,simplepo_order_ref],
 [christmas_simplepo_item0_ref,simplepo_item_ref],
 [christmas_simplepo_item1_ref,simplepo_item_ref]
 ]).
```

Table 3.37 – Formalization of references, their resolution and encoding in the context of traceability of a conformance relationship.

3.2.14 Additional Reasoning

So far, we have provided a set of axioms that form a static semantics of (language-centric) megamodels. To be more precise, we have formalized integrity constraints as formal implications. So far, we have use the symbol $\Rightarrow$ in any integrity constraint.

When creating a software model, elision and abstraction are applied. Thus, we aim for conciseness. For example, when we state that something is a member of a language, as in *elementOf("SimplePo.ecore", ecorexmi)*, we can already infer that *"SimplePo.ecore"* refers to an artifact. Hence, we do not need to explicitly add a fact to our knowledge base that *"SimplePo.ecore"* is an artifact. The following axioms focus on such inference of facts, where we use the symbol '$\Leftarrow$' to better distinguish such rules from integrity constraints as shown in Table 3.38. Every inference axiom has been realized in the Prolog implementation as well and is used in the tests against integrity constraints.[16] For testing purposes, an additional example on the web application framework Django has been added in the Prolog code, but we omit showing the code here to focus on the technology-independent theoretical aspects.

Inference	**Constraint**
$Artifact(a) \Leftarrow \exists l.elementOf(a, l).$	$Artifact(a) \Rightarrow \exists l.elementOf(a, l).$

Table 3.38 – Complete formalization concerned with constraint and inference.

Reflexivity and Transitivity

By inferring facts, we can ommit explicitly stating facts that can be inferred based on reflexitivity or transitivity as properties of a relation, for example, in a Prolog knowledge base. *partOfT* is a reflexive, antisymmetric and transitive relation. Thus, any implementation that wants to make use of the presented axioms may infer facts. While Table 3.39 only shows inference rules for *partOfT*, the rules for *subsetOfT* can be found in the Prolog implementation. We also include a sort of transitivity for the relation *elementOfT*. If an artifact is a member of a language that is a subset of another language, then the artifact is a "transitive" member of the superset.

Inference
$partOfT(x, x) \Leftarrow (Artifact(x) \vee System(x)).$ $partOfT(x, x'') \Leftarrow \exists x'.partOf(x, x') \wedge partOfT(x', x'').$ $elementOfT(a, l) \Leftarrow elementOf(a, l).$ $elementOfT(a, l) \Leftarrow \exists s.elementOf(a, s) \wedge subsetOfT(s, l).$

Table 3.39 – Sketched axioms on inferring reflexivity and transitivity.

The Language Triangle

By combining the knowledge we have collected so far, we discuss a bigger picture on language membership. We extend the axiomatization for language membership as follows. Any language membership of an artifact can be validated or inferred by an additional law. There exist two conditions combined by $\vee$ as given in Table 3.40 for the relation *elementOf*.

Every function is implemented by a system. If an artifact is an input/output of a function, the system implementing the function is an acceptor for the artifact. This way, if there is no

[16] `https://github.com/softlang/megaaxioms/blob/main/src/solasote/inference.pl` — Requested March 31, 2022

Constraint
$elementOf(a, l) \Rightarrow$ $(\exists s.defines(s, l) \wedge conformsTo(a, s))$ $\quad //Option1$ $\vee$ $(\exists f, ds, rs.funType(f, (ds, rs))$ $\quad //Option2$ $\wedge \exists is, os.funApply(f, (is, os))$ $\wedge \exists n.l == (ds \cup rs)_n \wedge a == (is \cup os)_n).$

Table 3.40 – Sketched axioms on integrity of language membership.

schema defining the language is persisted, there may still be an acceptor that implements the schema and thus the language. Again, we can also transform the integrity constraint to propose two separate inference rules as given in Table 3.41.

Inference
$//Option1$ $elementOf(a, l) \Leftarrow \exists s.defines(s, l) \wedge conformsTo(a, s).$ $//Option2$ $elementOf(a, l) \Leftarrow \exists f, ds, rs.imageOf((ds, rs), f)$ $\wedge \exists is, os.funApply(f, (is, os))$ $\wedge \exists n.l == (ds \cup rs)_n \wedge a == (is \cup os)_n.$

Table 3.41 – Inferring language membership.

A conceptual illustration of the first option is presented in Figure 3.2. The first option relates to the artifact that defines the language. If the language is defined by an artifact that serves as a 'schema', then any member of the language needs to conform to it.

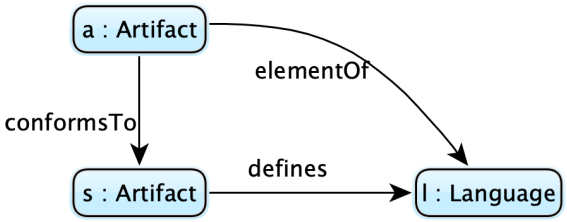

Figure 3.2 – Conceptual illustration on conformance-based validation.

A conceptual illustration of the second option is presented in Figure 3.3. The second option relates to implementation. If a language is implemented in terms of a function that accepts the language's members then any member of the language is a valid input of the function. Thus, they are potential inputs but may never become actual inputs of the function. Thus, there may still be language members that never take part in an actual modeled function application.

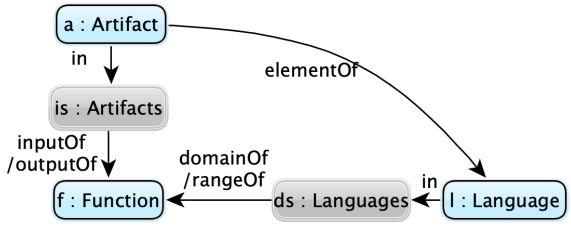

Figure 3.3 – Conceptual illustration on acceptance-based validation.

3.3 Conclusion

In this chapter, we have proposed a core ontology in terms of laws for language-centric megamodels [Favre et al., 2012c, Härtel et al., 2017], which is influenced by core vocabulary identified in a literature study. We have validated consistency and soundness through an implementation in Prolog that includes a knowledge base which models aspects of EMF usage. As a contribution to the megamodeling domain, this axiomatization can be seen as a static semantics composed of of integrity constraints and inference rules for implementing a reasoner. This approach can help with authoring as well as understanding megamodels. Many more case studies are needed to evaluate and potentially improve the core ontology.

We emphasize the importance of formalizing laws for relations by discussing how the problem with transitive part-hood raised in [Guizzardi, 2009] is solved in the core ontology SoLaSoTe. In the example by Giuzzardi, there are two different kinds of part-hood. First, a city (Rio de Janeiro) is a part of a country (Brazil), which is a property in a locality dimension. Second, a state (Brazil) is a part of an organization (UN), which is a property in a "membership" dimension. In our core ontology, we carefully differentiate between analogous part-hood dimensions. A fragment (*SimplePO.ecore/PurchaseOrder*) is part of a file (*SimplePO.ecore*). This is a property in the location dimension. The file (*SimplePO.ecore*) is a part of a language (*EcoreXMI*) since we interpret a language as a set of accepted members. That is a property in the "membership" dimension. In the example by Giuzzardi, we cannot infer that a city (Rio de Janeiro) is a member of an organization (UN), just because its state is a member. In analogy, we cannot infer that the fragment (*SimplePO.ecore/PurchaseOrder*) is a member of the language (*EcoreXMI*). In both cases, we have a kind of part-hood that relates to a location dimension and another that relates to a membership dimension. Through our formalization we have explained the meaning of the membership relation *elementOf* through additional constraints that involve the common relation of conformance to a specification as well as a function, where the member is either in the input or in the output. Through rigorous formalization and explanations, we aim to help software engineers with understanding common vocabulary in software documentation that focuses on the use of software languages and software technologies in SoLaSoTe.

Chapter 4

Software Languages at Wikipedia

Wikipedia is a rich source of information across many domains including software languages and software technologies. Yet, recovering articles relevant to a specific domain is a difficult problem since such articles may be rare and tend to cover multiple topics.

In this chapter, we develop a methodology that aims to classify Wikipedia articles based on their relevance to a given domain class. The methodology is tailored to be used when relevant articles are rare, i.e., a problem commonly referred to as a rare class problem. The methodology uses semi-supervised machine learning to retrieve a decision tree classifier; instead of considering all of Wikipedia relevant categories are identified; several content features of articles are considered including URL pattern, hypernyms recovered from the first sentence, lemmas in the summary, infoboxes and links from list articles; initially identified seed articles serve as representative positive training data.

In a case study, we classify articles by their relevance to the domain class of software languages. As a result, we identify features of an article that indicate its relevance to software languages based on 2897 positively and 98744 negatively classified articles in the category *Formal languages* as well as 745 positively and 5371 negatively classified articles in the category *Computer file formats*. By using 301 seed articles, we identify another 2797 articles on software languages. The classifier thoroughly evaluated through a survey, in which 31 domain experts participated.

Chapter contribution: The methodology and the case study for discovering relevant articles on software languages on Wikipedia is published in [Heinz et al., 2019]. This chapter elaborates on the published contribution in three directions. First, we present types of content, i.e., title, infobox, summary and lists, in articles on Wikipedia as knowledge resources. Second, we explain the barriers that come with the author culture established through guidelines. Third, we describe what software language classification dimensions are encoded by Wikipedia categories.

4.1 Knowledge Discovery on Wikipedia

Wikipedia is a very large-scale, continuous community effort to collect, organize and share knowledge in almost all domains. In general, the author guidelines and quality assessment challenge automated procedures with the goal to extract and structure knowledge based on Wikipedia [Stvilia et al., 2008, Do and Roth, 2012, Wu and Weld, 2008, Rebele et al., 2016].

Every article focuses on a main topic, e.g., the programming language Java. Recovering articles describing given topics in a specific domain is a reoccurring problem [Wang et al., 2010, Dong et al., 2016, Chen et al., 2016, Nassif et al., 2020]. We contribute to the research area of extracting knowledge on software concepts, specifically software languages [Lämmel et al., 2013a]. The specific case study on software languages and the methodology utilize several types of content that are turned into features for article classification. Throughout this section, we answer the following initial research question:

> *RQ W0: What types of content in a single Wikipedia article can be used for extracting knowledge and what challenges need to be addressed?*

4.1.1 Content Types

A Wikipedia article has structured and unstructured content. Here, we provide an overview based on illustrative examples. The covered content types are *Talk Pages*, *URL tags*, *Article summary*, *Full text*, *Article links*, *Category links*, *External links*, *Infoboxes*, and *Tables*. For each content type, we mention related work that uses it and provide insights into related Wikipedia author guidelines.

In the case study on software languages [Heinz et al., 2019], we access the content through *DBpedia* [Bizer et al., 2009]. *DBpedia* is an elaborate knowledge graph that is based on cleansed Wikipedia content. A subset of the unstructured content from Wikipedia is already turned into a usable feature for classification. We also explain in how far *DBpedia* preprocesses content and makes it easily accessible through its SPARQL endpoint.[1]

URL Tags

The URL identifying the article may contain additional terms for disambiguation. Many times, such additional term is added as a tag in parantheses. This way, many titles of Wikipedia articles encode a taxonomic classification. For example, the URL identifying the article on the programming language Java contains ‘programming language’ as an additional tag to provide disambiguation from the island or the platform. A part of the disambiguation page for Java is presented in Figure 4.1. It also includes ‘Java Virtual Machine’ which exemplifies taxonomic information even without a tag. The title can be used to identify a subclass relationship to “Virtual Machine” and semantic relatedness to “Java”.

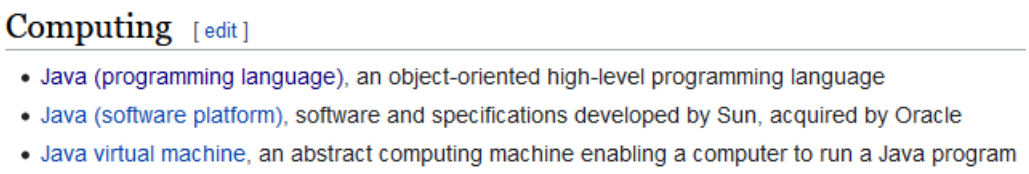

Figure 4.1 – Part of the disambiguation page for ‘Java’. Here, different additional tags hint at the class of the main topic such as ‘programming language’ or ‘platform’.

Hence, the URL tag may provide an additional term that can be used to identify a context or even initial bits of a classification. Its usefulness has been shown for named entity

[1] `http://dbpedia.org/sparql` — Requested March 31, 2022

recognition (NER) in *DBpedia Spotlight* [Mendes et al., 2011], which is a web tool for NER in text, named entity disambiguation in [Cucerzan, 2007], and taxonomic relation extraction in [Zarrad et al., 2013].

Article Summary

The summary, i.e., the leading text before the table of content in an article, provides the most central facts on the main topic.[2] For example, an instance of a programming language can be classified in several common dimensions such as its paradigm and abstraction level based on the first sentence. This is exemplified in Figure 4.2, where we provide an illustrative excerpt from the summary of the article on Java.

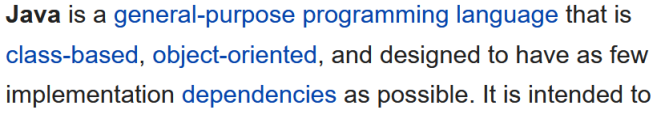

Figure 4.2 – Exemplary feature of an article: The Summary.

To use NLP techniques, we first need to remove the markup, e.g., internal links and infobox delimiters. Such cleaning can be complex as Wikipedia Markup is context-sensitive [Zaytsev, 2011]. Curated text that has already been preprocessed is provided by Wikimedia in Cirrus dumps online[3], which are meant to be used for searching facilities.[4]

DBpedia maintains the property *dbo:abstract*, where DBpedia pages contain a clean textual summary recovered from Wikipedia. The semantic relatedness of two terms can be measured based on the summary [Zhao et al., 2014, Frikh et al., 2011]. Following [Mirylenka et al., 2015], multiple kinds of relations can be extracted from the first sentence. Hypernyms provide a reliable feature as most articles on Wikipedia begin with a sentence containing 'is a' [Flati et al., 2014, Flati et al., 2016]. A WordNet synset can be matched with words in the summary in [Boinski et al., 2019] to connect the most suitable Wikipedia article with it.

To conclude the use of an article's summary, we point out technical details that were used when mining hypernyms from the first sentence in [Heinz et al., 2019], where we are interested in identifying articles on instances of a class, that are inspired by hypernym extraction as discussed in [Flati et al., 2014]. We extract the hypernym, if the sentence does not start with 'A'. If we find subset relationships as in 'A programming language is a formal language[...]', no hypernym is considered as a feature, because we want to focus on instances of a type instead of a subset relation. To achieve a high coverage, we also infer the hypernym from recovered membership relationships as in 'Perl 6 (also known as Raku[5]) is a member of the Perl family of programming languages', where 'languages' is the extracted hypernym. Actually, we implemented the extraction of instance-of relationships using multiple natural language pattern.[5] An example of an extraction for Java is presented in Figure 4.3. We also do not consider the subject of the sentence. By not recovering the subject of a sentence, we avoid having to perform named entity recognition and a dedicated resolution to an aligned entity. We abuse the fact that the first sentence of an article always gives a definition of the main topic.

[2] `https://en.wikipedia.org/wiki/Wikipedia:Summary_style` — Requested March 31, 2022
[3] `ftp://ftp.acc.umu.se/mirror/wikimedia.org/other/cirrussearch` — Requested March 31, 2022
[4] `https://www.mediawiki.org/wiki/Extension:CirrusSearch` — Requested March 31, 2022
[5] `https://github.com/softlang/wikionto/blob/master/src/features/cop_semgrex.py` — Requested March 31, 2022

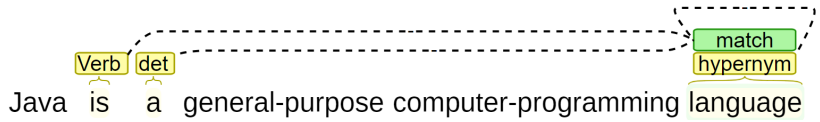

Figure 4.3 – Exemplified lexico-syntactic pattern for hypernym recovery.

Full Text

In the remaining text of a Wikipedia article that follows the table of contents, additional information is provided on the main topic, such as a detailed description of a topic's relationships to other topics or a historical timeline on the evolution of a concept. The text in Figure 4.4 illustrates how the subsection of the article 'Java (programming language)' provides more details that are not central to the concept itself but describe the relation between software development for Android mobile applications and the Java language. The full text is not available on DBpedia.

Android [edit]

The Java language is a key pillar in Android, an open source mobile operating system. Although Android, built on the Linux kernel, is written largely in C, the Android SDK uses the Java language as the basis for Android applications but does not use any of its standard GUI, SE, ME or other established Java standards.[70] The bytecode

Figure 4.4 – A subsection in the full text of 'Java (programming language)'. More detailed and less summarizing information is given.

Discovering and extracting knowledge requires the use of elaborate techniques from the domain of *Natural Language Processing*. The text following the table of contents may also explain other topics than the main topic. It may explain side topics that are just easier to explain in the same context.[6] Hence, NLP-techniques are necessary that are fit to be used for free text [Fairchild et al., 2015, Nguyen et al., 2007, Wu and Weld, 2010, Wang et al., 2010]. A more dedicated and general overview on processing natural language text is given in Section 7.6.

Article Links

The raw markup of any article also contains internal links to other articles, which is exemplified by the words highlighted in blue in Figures 4.2 and Figure 4.4.

In the methodology as published in [Heinz et al., 2019], we consider Wikipedia list articles. A list article contains names of and potentially links to entities that are related to the topic encoded by the list article's title. The title provides additional information, e.g., 'List of dog breeds'. We consider a link from a list to an article as a feature. We did not consider links from an article to any other article as it turned out to not introduce any discriminant feature and introduced technical challenges by increasing the size of the feature matrix.

Links in general as well as links from list articles have been investigated in research. Internal links from list articles are used to infer types of articles in [Kuhn et al., 2016]. An ontological relationship is recovered based on the link in [Völkel et al., 2006], where the authors aim to create a semantically rich article network. In [Dong et al., 2016], subsumptions are learned from articles describing domain concepts based on Hearst pattern. In a related

[6] `https://en.wikipedia.org/wiki/Wikipedia:Notability#Whether_to_create_standalone_pages` — Requested March 31, 2022

approach [Chen et al., 2016], the concept set is learned by matching articles with Stackoverflow tags. Based on the article- and category graph, the concept set is expanded and further relationships are learned.

Category Links

As a special type of link, articles are grouped based on links to categories. According to Wikipedia's guidelines, there exist two types of categories, namely topic categories[7] and set categories.[8] While topic categories group all articles that relate to one topic such as *Opera*[9], set categories group their set members, e.g., *Operas*.[10] Any assignment of a category to an article needs to be verifiable.[11] Hence, information encoded in the article needs to hint at relatedness to a topic or set membership. Set membership can also be expressed through subcategorization, when the category is an eponymous category, i.e., a category that represents a topic of an article[12], e.g., the category *Java (programming language)*. Subcategorization can have different meanings. In both senses, topic and set, an is-A relation may be encoded by subcategorization, i.e., subset relation for sets and subtype or instance-of relation for topics. Otherwise, only relatedness may be encoded[13].

In the DBpedia knowledge graph, category assignment is mapped to the property *dct:subject*, which is introduced as a relation from the *dublin core ontology*.[14] Subcategorization is mapped to *skos:broader* as a relation from the *SKOS ontology*.[15]

An example of categories assigned to the article on *Java (programming language)* is given in Figure 4.5. It exemplifies set membership as in 'Class-based programming languages', relatedness as in 'Java platform', and the assignment of the eponymous category 'Java (programming language)'.

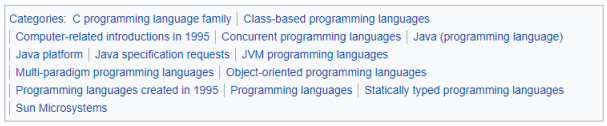

Figure 4.5 – The set of categories assigned to Java. Assignments show set membership as well as topic relatedness. Every category needs to be verifiable through information in the article.

The category graph can be used for clustering documents [Hu et al., 2009], for computing semantic relatedness [Strube and Ponzetto, 2006] and for extracting a taxonomy [Wang et al., 2010, Flati et al., 2014, Flati et al., 2016], or more specifically, software language taxonomy [Lämmel et al., 2013a].

External Links

The article links to external resources in order to assure that information is verifiable. The links to external collections as a kind of bibliography can be found in dedicated sections with

[7] `https://en.wikipedia.org/wiki/Wikipedia:Categorization#Topic_categories` — Requested March 31, 2022

[8] `https://en.wikipedia.org/wiki/Wikipedia:Categorization#Set_categories` — Requested March 31, 2022

[9] `https://en.wikipedia.org/wiki/Category:Opera` — Requested March 31, 2022

[10] `https://en.wikipedia.org/wiki/Category:Operas` — Requested March 31, 2022

[11] `https://en.wikipedia.org/wiki/Wikipedia:Categorization#Categorizing_pages` — Requested March 31, 2022

[12] `https://en.wikipedia.org/wiki/Wikipedia:Categorization#Eponymous_categories` — Requested March 31, 2022

[13] `https://en.wikipedia.org/wiki/Wikipedia:FAQ/Categorization#What_is_the_purpose_of_categories?` — Requested March 31, 2022

[14] `https://asistdl.onlinelibrary.wiley.com/doi/full/10.1002/bult.70` — Requested March 31, 2022

[15] `https://www.w3.org/TR/2008/WD-skos-reference-20080829/skos.html` — Requested March 31, 2022

the title ‘Notes’, ‘External links’, or ‘Further reading’.

In [Tzekou et al., 2011], the quality of external links in Wikipedia articles is assessed, for example, by measuring frequency of domain name spaces and decay of links. The authors also discuss implications on the quality of Wikipedia overall. In [Kaptein et al., 2010], whether web entity retrieval can be improved by using Wikipedia as a pivot is investigated. There, a Wikipedia article is matched based on an already available external link and then used recover additional web links from the article.

Infobox

The infobox summarizes information for a specific instantiated concept, such as programming language. For common concepts, dedicated templates are supposed to be used for the purpose of consistency and standardization.[16] It is encoded in terms of raw markup. It can be used to discover an initial set of ontological properties. The examplary infobox for Java is provided in Figure 4.6. In the infobox, we see classifying properties, such as paradigm, and typing discipline as well as relatedness to other topics, such as its designer, and influencing and influenced languages.

Figure 4.6 – The exemplary infobox for ‘Java (programming language)’. It contains classification properties in several dimensions as well as relationships to other languages.

The properties in the infobox are mapped to ontological properties in DBpedia. When we inspect the page for Java[17], we find relationships to other resources as in *dbo:influenced* as well as reversed relations from other resources as in *is dbo:influenced of*. We also see that the namespace *dbo* relates to properly resolved ontological properties, while *dbp* only seems to represent some fields from the infobox, e.g., *dbp:typing* which at least links to other ontological resources, and *dbp:paradigm* which is a textual property where the object contains the text ‘Multi-paradigm: Object-oriented , structured, imperative, generic, reflective, concurrent’.

[16] `https://en.wikipedia.org/wiki/Wikipedia:Manual_of_Style/Infoboxes` — Requested March 31, 2022

[17] `http://dbpedia.org/page/Java_(programming_language)` — Requested March 31, 2022

In [Wu and Weld, 2008], infoboxes are improved with regard to their coverage of articles by processing text and using supervised learning. In [Aprosio et al., 2013], the coverage of DBpedia properties is improved by processing natural language text using distant supervision. In [Biswas et al., 2018], an article's table of content, its abstract, and recognized named entities in the abstract are used to predict the infobox template. In [Morales et al., 2016], infobox information is used for question answering.

Tables

Several articles provide comparisons of concepts within the same domain. Such comparisons are often structured through tables. A table is an enhanced list. According to guidelines any table should facilitate to better understand and compare an overview or comparison of similar items[18]. Figure 4.7 depicts an example of a table from the article on 'Comparison of parser generators'.[19] Every column in the table can be used to sort the rows; names as well as properties are linked; links as well as properties are incomplete.

Name	Parsing algorithm	Input grammar notation
ANTLR4	ALL(*)[2]	EBNF
BtYacc	Backtracking Bottom-up	?
Kelbt	Backtracking LALR(1)	?
LPG	Backtracking LALR(k)	?

Figure 4.7 – A snippet from 'Comparison of parser generators'. The table for parser generators provides exploration facilities for a quick overview over properties such as the name, algorithm and notation.

Research deals with the feasibility of recovering knowledge from tables. In [Sannier et al., 2013], tables in the context of comparisons of similar items are considered as product comparison matrices. An empirical investigation shows that such tables suffer from ambiguity and a lack of formalization. In [Cannaviccio et al., 2018], knowledge graphs are fed by processing data in tables. In [Fetahu et al., 2019], a tool is proposed that recovers relations between tables on Wikipedia.

Talk Pages

Wikipedia articles' quality is communicated on dedicated talk pages via grades. While summarizing findings from [Stvilia et al., 2008], we refer to and check against the status quo published in terms of official information by Wikimedia. One can find the grade of an article on its talk page. The grades follow an official scheme that ranges from a featured article over several intermediate grades to a stub article.[20] The grade *Featured* represents the highest level of quality according to quality criteria such as accuracy, neutrality, completeness, and style.[21] Featured articles are supposed to adhere to referential quality standards. The same principle is applied to list pages according to usefulness, completeness, accuracy, neutrality, style and prose. It is noteworthy that list articles thereby are treated as a special type of article.[22] At last, Wikimedia provides a description of what makes a perfect article.[23] Standard quality

[18] `https://en.wikipedia.org/wiki/Wikipedia:Manual_of_Style/Tables` — Requested March 31, 2022
[19] `https://en.wikipedia.org/wiki/Comparison_of_parser_generators` — Requested March 31, 2022
[20] `https://en.wikipedia.org/wiki/Template:Grading_scheme` — Requested March 31, 2022
[21] `https://en.wikipedia.org/wiki/Wikipedia:Featured_articles` — Requested March 31, 2022
[22] `https://en.wikipedia.org/wiki/Wikipedia:Featured_lists` — Requested March 31, 2022
[23] `https://en.wikipedia.org/wiki/Wikipedia:The_perfect_article` — Requested March 31, 2022

measures are included such as notability, clarity, understandability, precision, explicitness and verifiability. Research with the goal to evaluate quality of articles in Wikipedia typically predicts (a subset of) mentioned grades based on structural features such as length, and social features such as the number of contributors [Bassani and Viviani, 2019a, Bassani and Viviani, 2019b].

4.1.2 Challenges

To effectively use information on Wikipedia, several challenges need to be known in advance. They affect multiple if not all content types of articles.

Multiple topics

An important challenge is that, in a large number of cases, *a single Wikipedia article covers multiple topics*. Looking at the very first sentences of the article about MATLAB (see Figure 4.8), several topics are mixed together (e.g., user interfaces, numerical computing, and software languages). The category graph is impacted as well (e.g., linear algebra and array-programming languages are both mentioned). There are also two infoboxes—one about MATLAB the *language*, the other about MATLAB the *software*. For illustration, we provide an overview in Figure 4.8, where each type of feature is marked.

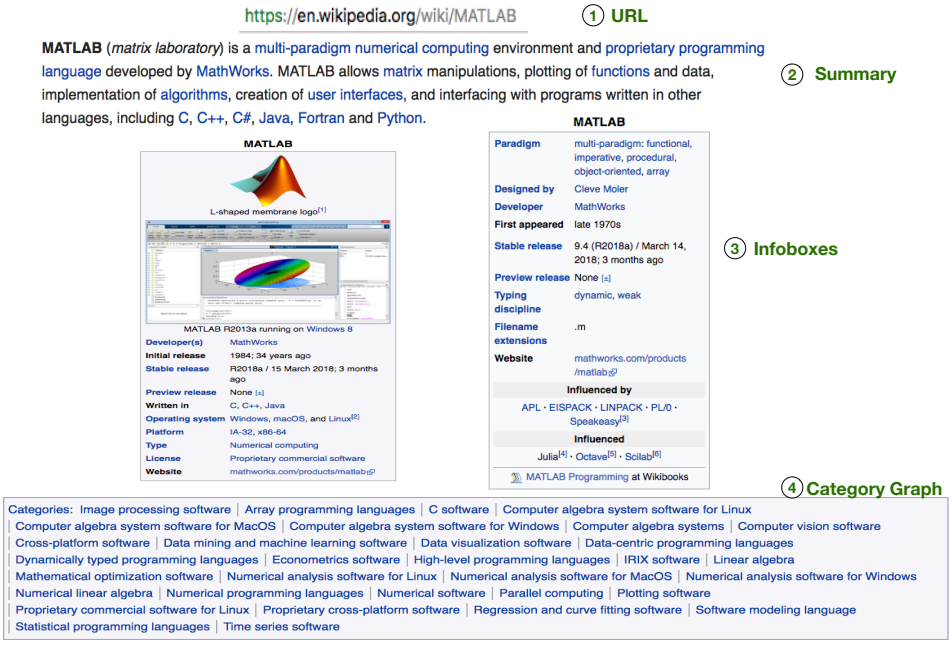

Figure 4.8 – General structure of a Wikipedia article and some indicators.

Many languages and their implementing software, e.g., 'MATLAB', are discussed in the same article. Such articles then may contain an infobox with template 'software' and another infobox with the template 'programming language' at the same time. The used infobox template is not always useful as well. The article 'OpenDocument' uses the abstract template 'Infobox', so that name, value pairs are added without any constraints.

Notability

In the guidelines, the subsection on notability[24] states that authors should explain multiple topics in a single article, when this benefits understandability. The cultural requirement for any article to be on a notable topic can become a challenge for automatically extracting domain-specific knowledge, because not every article represents a single concept, such as a software language [Lämmel et al., 2013a, Heinz et al., 2019] or software technology [Nassif et al., 2020]. Relevant concepts may be hidden in articles on related concepts. As a result, it leads to multiple topics explained in a single article. While this guideline aims to improve understandability, it poses a challenge to automatically reusing the knowledge that is encoded by an article.

Lack of Scientific Classification

For some domains, like animals [Wang et al., 2010], categories that represent scientific classification are consistently used and provide a decision ground for still identifying relevant articles. In the domain of software languages either such crucial indicators do not exist for articles or they are used inconsistently. The recovery of relevant articles in such domains becomes looking for needles in a hay stack.

Ambiguous Categorization

Wikipedia's category graph cannot be directly used for classification within the domain. This is confirmed as a problem, for example, in [Mirylenka et al., 2015]. Categories may serve purposes other than classification, for example, collecting articles related to a common topic, such as Java.[25] We still observe categories that do not follow naming conventions[26], e.g., the category 'High Integrity Programming Language'[27] uses the singular case while that would actually imply a topic category. Hence, topic and set categories cannot always be distinguished and properly diffusing sub-categorization poses even more challenges that need to be resolved.

Domain Breach

In initial explorations, we found out that the depth of the category tree, e.g., below 'Computer languages' seems to be infinite and as we go deeper many more non-software language articles are included. In Section 4.2, we propose to cut off the category tree at the depth of the deepest known seed article. Even though subcategorization loops should be avoided according to author guidelines[28], we still observed them.

DBpedia Quality Issues

When we extract facts on software languages from DBpedia, we need to deal with known and frequent kinds of quality issues. In [Acosta et al., 2018], the feasibility of using experts and crowd workers (Amazon mechanical turks) for detecting quality issues in DBpedia is evaluated. The performance of the experts and crowd workers is compared to that of a straightforward

[24] `https://en.wikipedia.org/wiki/Wikipedia:Notability#Whether_to_create_standalone_pages` — Requested March 31, 2022

[25] `https://en.wikipedia.org/wiki/Category:Java_(programming_language)` — Requested March 31, 2022

[26] `https://en.wikipedia.org/wiki/Wikipedia:Categorization#Naming_conventions` — Requested March 31, 2022

[27] `https://en.wikipedia.org/wiki/Category:High_Integrity_Programming_Language` — Requested March 31, 2022

[28] `https://en.wikipedia.org/wiki/Wikipedia:Categorization` — Requested March 31, 2022

baseline, where consistency issues are automatically recovered by the use of a framework. The results of the feasibility study show that a combination of both groups performs more precise in several exemplary cases. The correction of the detected quality issues by crowd workers is not studied as the authors assess it as not the most cost-efficient method. In a previous study, the authors have studied the severity of quality issues and different types of issues in DBpedia [Zaveri et al., 2013]. The feasibility study in [Acosta et al., 2018] only focuses on the three most frequent types of quality issues: *Incorrect/incomplete object:* objects of a triple may instantiate the wrong type; *Incorrect datatype or language tag:* labels at the object position of a triple may have an incorrect datatype or language tag; *Incorrect link:* two resources are wrongly associated. In early experiments on discovering knowledge on software languages in DBpedia, we were able to confirm quality issues. For example, the DBpedia page on the Java programming language[29] has a property *typing*, where a link to the kind of type system, such as a static type system, is assigned. Instead, the concept *type system* itself is given as a value. This may be caused by the automated translation from Wikipedia, where the Infobox of the article[30] contains the text 'Static, strong, safe', which is then linked to the article on type systems.

4.1.3 Significance of the Challenges

Here, we provide an analytical insight into the effects of the previously mentioned challenges for an exemplary root category. Thereby, we motivate why they need to be addressed when reusing Wikipedia as a knowledge resource. We initially explore articles that link to (subcategories of) the category 'Computer languages' and search for the top ten frequent nouns ('NN' tag recovered with part-of-speech tagger). We cut off the category tree at a depth of seven. Figure 4.9 reports the number of articles for the top ten most frequent nouns in the category tree of 'Computer languages'. In the category tree, we observed topics that belong to other domains, such as 'songs', 'stars', 'album' and 'music'. Except for the minority that describes domain specific languages in, for example, the music domain, most of the article are obviously irrelevant to software languages. Figure 4.9 provides evidence that irrelevant topics are frequently covered below the selected category.

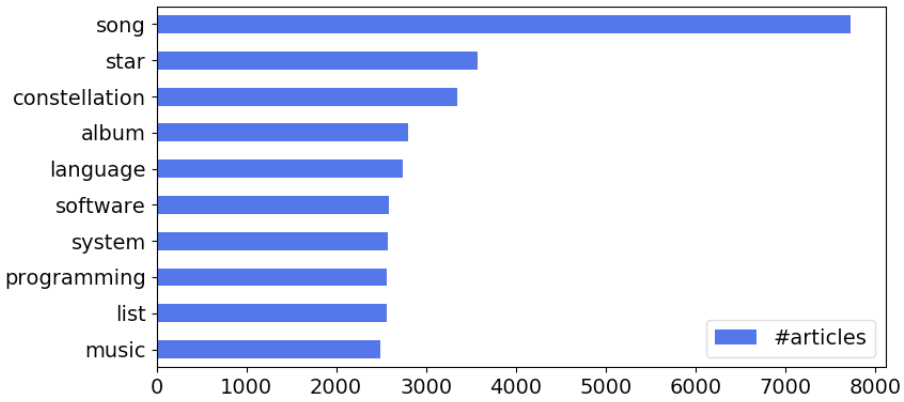

Figure 4.9 – Top 10 most frequent nouns in articles below 'Formal languages' category.

4.1.4 Related Approaches

For deriving domain ontologies from Wikipedia, various distinct approaches can be found [Metke-Jimenez et al., 2010]. The approaches range from support for manual crafting [Völkel et al.,

[29] `http://dbpedia.org/page/Java_(programming_language)` — Requested March 31, 2022
[30] `https://en.wikipedia.org/wiki/Java_(programming_language)` — Requested March 31, 2022

2006, Vrandecic, 2012] to unsupervised construction based on text [Sarencheh and Schiffauerova, 2017]. In [Wang et al., 2010], a domain ontology for animals is extracted from Wikipedia based on the category graph, article graph and section structure in articles. For the animals domain, most pages are maintained well and describe exactly one concept with a consistent section structure. In [Mirylenka et al., 2015] subset, membership, part-of, sub-topic and other relations are extracted by using the varying nature of subcategorization. Subsumptions from articles describing domain concepts based on Hearst pattern are recovered in [Dong et al., 2016]. In a related approach [Chen et al., 2016], the concept set is learned by matching articles with Stackoverflow tags. Based on the article- and category graph, the concept set is expanded and further relationships are learned. Related works discover knowledge from isolated structural features, such as the title [Zarrad et al., 2013, Mirylenka et al., 2015], text [Flati et al., 2014, Presutti et al., 2014], an article's section structure [Wang et al., 2010], links to other articles [Chen et al., 2016, Presutti et al., 2014], infoboxes [Wu and Weld, 2008], lists [Kuhn et al., 2016].

4.2 Seed-based Learning

We develop a *semi-supervised learning* methodology for identifying articles relevant to a domain-specific class for which only a limited seed can be leveraged. A seed is a set of highly representative articles for the domain class that have been identified by matching items from a trustworthy external source. In the case study presented in Section 4.3, we consider the lists of software languages on GitHub and TIOBE as seeds for discovering software language articles. Furthermore, we discuss the relevance of article features as indicators by inspecting a learned decision tree that include URL patterns, summary text, infoboxes, list articles as features and the category graph as a scope limiter. The developed methodology and the performed case study as described in the next section answer the question that has already been stated in [Heinz et al., 2019].

> *RQ W1: How can we classify Wikipedia articles by their relevance to a given domain when relevant articles are rare and multiple main topics are covered by articles?*

Figure 4.10 provides an overview of the methodology. i.) For training, we recover articles that match seed elements from trustworthy external resources. ii.) Based on such seed, we define the scope in which we want to isolate relevant articles and label randomly sampled articles from this scope for additional training data. iii.) From the training data, we build a feature matrix with categorical values that state whether a structural feature, such as the programming language infobox template, is present. iv.) We train a binary classifier that decides whether an article is relevant. When we explain each step, we already draw examples from the case study, but leave the detailed discussion of case study results to Section 4.3.

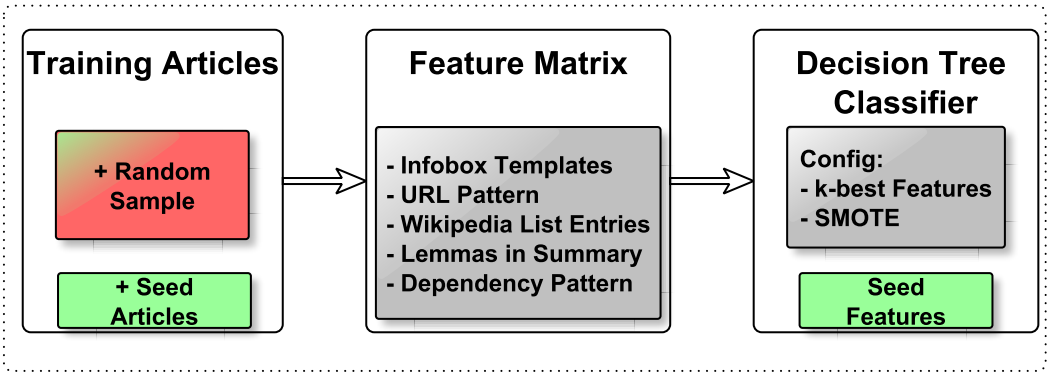

Figure 4.10 – The ingredients of the approach. The seed in the training data influences decisions as its common features are very decisive.

Training Articles

The set of training articles is the union of seed articles, which match the names from an external source (e.g., *GitHub* and *TIOBE*), and random articles that are sampled from a predefined scope. To explain this step in a more comprehensive manner, we anticipate results from the case study.

Seed Articles: Matching names from seeds with Wikipedia articles is challenging. We encountered favorable as well as problematic cases. To our favor, most names can be matched with article titles by leaving out tags (e.g. '(programming language)') or by using Wikipedia redirects. However, manual reviewing remains crucial to make sure that languages are matched correctly, e.g., 'Red' can either be matched with 'Red (programming language)' or color. In some cases, we succeeded by varying the writing style for the name, e.g., by unfolding

acronyms, such as EBNF. In less favorable cases, we propose to use Wikipedia's search engine to detect mentions in the text of existing articles. Only if the existing article describes a seed element as its main topic in the summary, we take it as the recalled article. To ensure availability of the necessary features for classification, we exclude stub articles from further analysis. At worst, a seed name is not even mentioned in any article.

Random Articles: Beyond representative relevant articles, we aim at manually labeling representative training data that also includes irrelevant articles. Therefore, we choose to randomly sample articles in a predefined scope. As previously described in Section 4.1, we have manually observed that the category graph poses several challenges. Thus, we cannot just consider every article transitively reachable by a *set category* as relevant. We need to deal with domain breaches (see Section 4.1.2).

We propose to randomly sample in a narrowed scope of articles. The scope is determined by manually selecting root categories so that all seed articles are contained in the scope based on (subcategory) links. Additionally, we propose to restrict the depth so that at least one seed article exists at the lowest level. This way, we intend to reduce the scope to avoid introducing overly many negatively classified articles while gaining only one or two positive articles at further depth levels.

Furthermore, we also exclude stub articles[31] from analysis, since they may not provide enough features as a decision ground for an automated decision on their relevance. As we consider list articles as features, we exclude them from random sampling as well.

Feature Matrix

Next, we describe which features are extracted from the content types discussed in Section 4.1. All features are of a categorical nature and state whether something is present or not. In order to reduce the number of features, we only consider features that are present in at least ten articles in the training data.

- *Infobox templates* are frequently used in articles. In the case study on software languages, it is one of the most decisive features, since infoboxes already serve as quick summaries of a structured classification.
- *URL tags* are included as separate features, since information on disambiguation is a trivial hint on a domain class.
- *Hypernyms* are extracted based on dependency pattern applied to the first sentence of an article's summary.
- *Lemmas* are extracted from the whole summary of an article excluding stop words. They turn out to be valuable indicators especially for isolating the software language articles from other domains.
- *Links from List Articles* are extracted in two steps. First, we recover all list articles in the scope of the root categories. Second, we encode whether an article is linked to a recovered list article as a categorical feature.

Decision Tree Classifier

Based on the built feature matrix, we learn a decision tree classifier. We chose this type of supervised learning technique in order to gain transparency in the decision process. This transparency allows us to debug the classifier as well as to find out what the best features are for automatically recognizing relevant articles.

[31] `https://en.wikipedia.org/wiki/Wikipedia:Stub` — Requested March 31, 2022

A rare class problem is challenging but resolvable by common techniques. To explicitly counter the rare class problem that may affect the learning process of a classifier, we use oversampling with *SMOTE* [Chawla et al., 2002] for synthesizing more training data to reestablish the balance between relevant and irrelevant articles in the training data. The underlying hypothesis for our approach is that the resulting decision tree mainly classifies based on features present in seed articles. Thus, seeds have a more general interest: *Seed articles provide representative features for recognizing relevant articles.* Metaphorically speaking, they isolate relevant articles from the rest. In the case study, we show which features of seed articles provide a representative positive indication and hence may be found in a fit decision tree.

Evaluation

We investigate the results from evaluating a classifier concerning its accuracy based on labels by experts. Here, we train a decision tree classifier so that we can present technical insights on how decisions are made and explain the effects of using a representative seed in more detail. As a result, we conclude with presenting how many relevant articles and categories are estimated in a limited scope.

4.3 Case Study on Software Languages

We developed a seed-based learning methodology for identifying articles relevant to a domain-specific class while leveraging an available limited ground truth. For the domain of software languages, the seed is based on two commonly known lists, namely, the list of languages recognized by Github and the popularity index TIOBE. We identify the indicators for relevant Wikipedia articles by inspecting a learned decision tree that include URL patterns, summary text, infoboxes, list articles and category graph. We present a case study which, in itself, results in the most comprehensive corpus on software languages available today. The results are evaluated by domain experts through a survey. The datasets including plotted decision trees and survey data are available online.[32]

For the case study, we identify the competencies of the ontological class *software language*. The first explanation on what a software language is can be drawn from multiple literature resources: A software language is any artificial language that is used in the development of software [Lämmel, 2018, Favre et al., 2009, Kleppe, 2008]. We derive competencies from [Favre et al., 2009, Lämmel, 2018] and explain them based on examples.

- *A software language is a set of digital artifacts.* Hence, it serves as a classifier of its members. Most files with the '.java' file ending are elements of the Java language. Competencies hence directly relate to its members.
- *The syntax is formally defined.* This is typically covered by grammars and language specifications. For example, there exists an informal specification by the *Internet Engineering Task Force (IETF)* that contains an *ABNF* grammar for *CSV*[33].
- *Well-formedness rules are (in-)formally defined.* For any programming language, such well-formedness is defined using a type system. For the UML, the specification contains OCL constraints[34]. For CSV, constraints are informally defined in the specification, e.g., 'Each line should contain the same number of fields throughout the file.'
- *The semantics is (in-)formally defined.* Interpreters define execution semantics and translators (i.e., compilers) provide a translational semantics, e.g., for DSLs [Schauss et al., 2017]. For representation formats like CSV or XML, the meaning of syntactic constructs relates to where the entry of the next element or attribute begins.
- *The pragmatics is (in-)formally defined.* For CSV, the specification elaborates on pragmatic issues such as 'Security considerations'. It also defines its pragmatics in terms of its purpose 'The comma separated values format (CSV) has been used for exchanging and converting data between various spreadsheet programs for quite some time.'

4.3.1 Executing Seed-based Learning

We execute the methodology from Section 4.2 for the domain of software languages.

Training Articles

As the first step, we have identified and reused two trustworthy external data sets: both act as *seeds* for recognizing relevant articles. *GitHub* presents statistics on which languages are used for any repository. The complete list of languages that are recognized can be extracted[35].

[32] `https://github.com/softlang/wikionto/` — Requested March 31, 2022
[33] `https://tools.ietf.org/html/rfc4180#page-5` — Requested March 31, 2022
[34] `https://www.omg.org/spec/UML/About-UML/` — Requested March 31, 2022
[35] `https://github.com/github/linguist/blob/master/lib/linguist/languages.yml` — Requested March 31, 2022

The *TIOBE* index presents statistics on how often software languages (mainly programming languages) are mentioned on the web. Here, the list includes variations of names[36]. We match the names from the lists with articles on Wikipedia. The resulting statistics from matching articles are given in Table 4.1. As the union of TIOBE and GitHub, 327 Wikipedia articles match a seed from at least one source. We observe that GitHub's list contains more elements and 108 Github languages as well as only 9 TIOBE elements cannot be recognized in an article. Additionally, 150 GitHub elements and 38 TIOBE elements are only mentioned in the context of an article, which cannot be considered as a part of the seed articles.

	GitHub	**TIOBE**
Matched Seed	241	206
Mentioned Seed	150	38
Unrecognized Seed	108	9
Total Seed	499	253

Table 4.1 – The summary for matching seed articles with how many seeds were matched, casually mentioned or remain unrecognized.

Next, we determine a restricted scope based on transitive containment in manually identified root categories and randomly sample 4000 articles. We assumed that Wikipedia's notion of "computer language" – a notion that is used all across computer science and beyond – is essentially equivalent to the notion of "software language"; see also [Favre et al., 2009] for a discussion. Hence, we hypothesized that all relevant languages link to "Computer languages", but we found "Augmented Backus–Naur form" as a seed member that links to 'Formal languages', where 'Computer languages' is a subcategory. Moreover, for seed articles such as 'CSV', we noticed the upper category 'Computer file formats' that is disconnected from 'Formal languages'. Such formats are categorized differently, but they do conform to the definition of software language. Hence, our experiments show that all seed articles are linked to (subcategories of) two disconnected roots in the category tree at a maximum depth of eight. Table 4.2 summarizes the initial depth-wise seed article frequency. In the limited scope, we exclude articles at a distance to the upper categories that is higher than 8.

	0	1	2	3	4	5	6-8
Formal languages	$\frac{2}{165}$	$\frac{6}{152}$	$\frac{85}{643}$	$\frac{136}{1494}$	$\frac{70}{3032}$	$\frac{16}{5141}$	$\frac{7}{91014}$
Computer file formats	$\frac{8}{228}$	$\frac{22}{511}$	$\frac{11}{529}$	$\frac{7}{764}$	$\frac{0}{574}$	$\frac{0}{339}$	$\frac{3}{3171}$

Table 4.2 – Number of seed articles per depth in the nominator. Number of articles from which we take training data in the denominator.

Feature Matrix

As a resulting dataset, the extraction returns a feature-matrix with 104,186 articles in total (includes data not used for training) and 46,173 features as the sum of all feature types. Every feature is categorical and hence has the label '1' for present and '0' otherwise. The set of training articles consists of the seed and 4,000 randomly sampled articles that were manually labeled. To avoid memory issues and enable loading the whole article-feature matrix, we use sparse matrices. For the seed, we report on the frequency of common features.

[36] `https://www.tiobe.com/tiobe-index/programming-languages-definition/#instances` — Requested March 31, 2022

- *Infobox Template:* In the scope, there exist 864 distinct infobox template names. 263 seed articles use an infobox template. We found the templates on 'programming language' (215), 'file format' (32), 'technology standard' (3) and 'software license' (1). Especially, the templates 'programming language' and 'file format' provide a strong positive indication, but they do not exist for every relevant article.
- *URL Pattern:* We extract all words separately in braces from every article's title as they provide a semantic annotation for disambiguation. Only around a third of the seed articles has such an annotation. The words in braces appearing more than once with their frequency are: 'language' (125), 'programming' (114), 'software' (6), 'stylesheet' (3) and 'markup' (2). The word 'programming' always appears together with 'language'.
- *Hypernyms:* In the seed, we found 'language' (235), 'format' (16) and 'dialect' (10) as the most frequent hypernyms. The hypernym 'language' and 'dialect' are also recovered when analyzing articles on natural languages.
- *Lemmas:* We recovered 5449 lemmas from seed articles while 457,164 distinct lemmas can be recovered from all articles in the scope. We enumerate the top five lemmas from the seed: 'language' (301), 'programming' (253), 'use' (216), 'develop' (120), 'design' (117). Lemmas are often used for topic models [Sarencheh and Schiffauerova, 2017].We found that they are essential to distinguish the domain of software languages from other unrelated domains such as natural languages.
- *Links from List Articles:* In the seed, we found 267 articles linked in such lists, such as the 'List of programming languages'. Not all mentions of software languages in lists may be recovered, because there are entities named as part of a list with a missing link.

Decision Tree Classifier

While we report one single configuration in [Heinz et al., 2019], multiple configuration parameters can be explored. Such parameters include i.) the *number of random samples* in the training set, ii.) the value for k in the *k-best feature selection*, iii.) whether oversampling is used, iv.) whether undersampling is used. The decision tree resulting from training with 4,000 random samples and $k = 23$ and the use of oversampling can be viewed online[37]. Next, we evaluate the results. We repeat the answered research question at the beginning of a subsection.

4.3.2 Evaluation - Precision & Recall

RQ W2.1: How well does the classifier perform beyond the seed?

To gain an evaluation set that is as objective as possible, we conducted a survey in which 31 domain experts from various research groups participated. As depicted in Figure 4.11, in each question in the survey, participants decided whether a presented article explicitly describes a software language as its primary topic.

We received 990 articles labeled by at least two experts. In order to gain additional insights on problems with decision-making, we emphasized the possibility of commenting on each question. For 43 articles, experts did not agree with each other. We identified following border cases, for which experts do not agree with each other.

- Experts may or may not refer to *logic systems* as a software language. For 'Noise-based logic', we received the following comment: 'Given that it is a logic, syntax and semantics can be defined - probably also the type system'.

[37] `https://github.com/softlang/wikionto/blob/seke19/data/datasets/sltreeKBest_23_mutual_info_classif_Oversampling_name.pdf` — Requested March 31, 2022

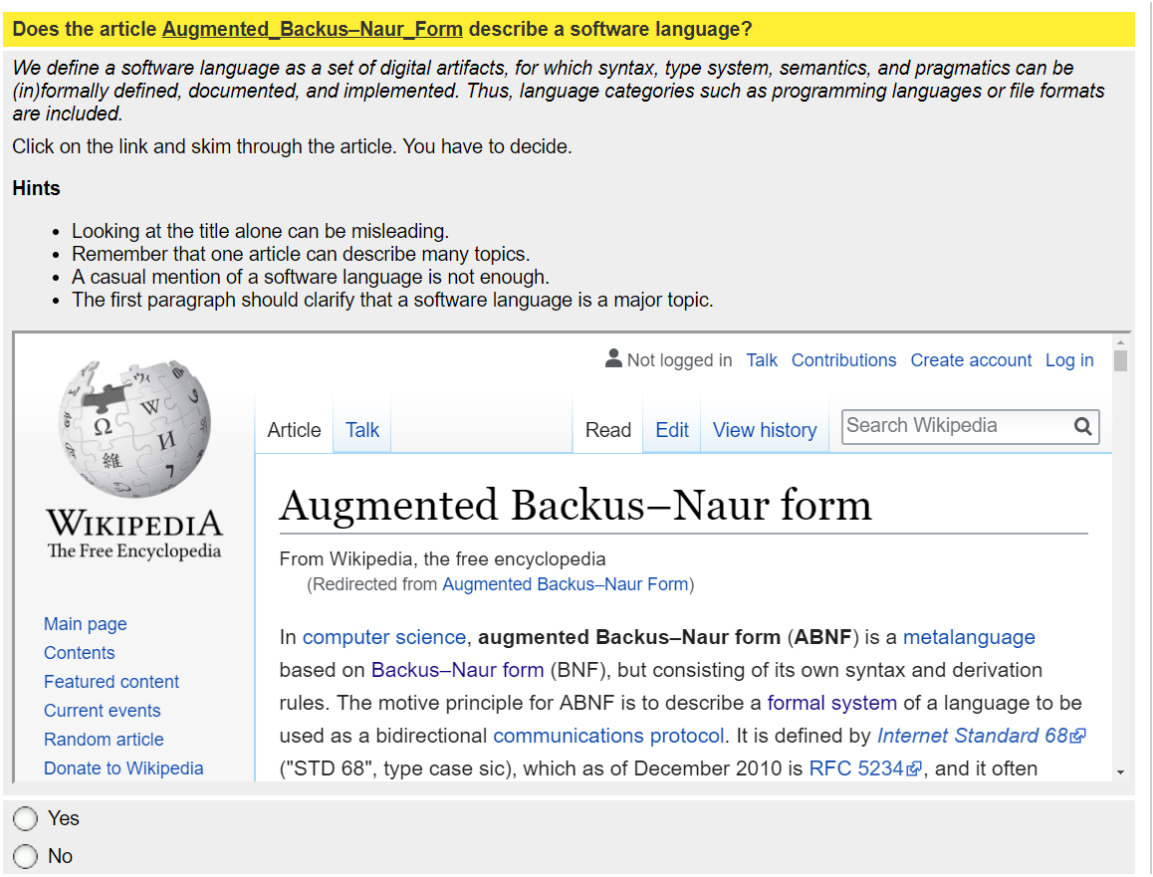

Figure 4.11 – An exemplary page in the survey, where 31 experts were asked to label articles on their relevance.

- *Frameworks* may be considered as software language as well. For 'JUMP GIS', we received the following comment: 'Frameworks can be software languages – depending on where you put the bar'. This way, the need for more precisely defined competencies is requested that need to be understood before labeling.

- *Derivates of OWL* exemplified by 'OWL-S', may be considered as software languages. Actually both experts, the expert labeling it as positive and the expert labeling it as negative commented. The negative label is commented as follows: 'I think it is a language as it is built on top of OWL, but it is not said explicitly in the article.' The comment for the positive label then expresses insecurity 'As an ontology strictly speaking a software language with type system and semantics etc., not sure if this is what you are looking for.'.

- *Query engine* articles may also relate to a specific software language. We again received two comments on 'Presto (SQL query engine)'. The positive label is commented as follows: 'It depends of the definition of a language. But if one sees presto as an SQL interpreter, it could be categorized as a dialect of SQL, hence a software language'. The comment for the negative label again refers to the fact that the article does not provide recognizable features: 'SQL-Dialect not mentioned in Article'.

As long as experts cannot reliably decide for an article whether a software language is described, a machine cannot as well. As a consequence, we decided to exclude articles, for which the experts did not agree. Since only ~4% is excluded, the threat to validity is reasonably low. We consider the remaining expert labels as the evaluation set.

With $k = 23$, the learned classifier performs with an *f1-score* of 0.7, *balanced accuracy* of 0.9, *recall* of 0.81 and *specificity* of 0.99.

The performance results for using 23 features are reasonably good. We recall 81% of software language articles and recall 99% of irrelevant articles as the specificity. Hence, most of the irrelevant articles are filtered out. The balanced accuracy, which already takes the imbalance of labels into consideration, as well as the f1-score show that there are still difficult cases, where the decision tree computes a false class. When trying to collect all software language articles in the scope, the decision tree helps, but additional work is left to identify and clean up false positives as well as false negatives.

For the future of SLEBOK [Combemale et al., 2018], we emphasize the need of a further investigation to truly understand the borders of the ontological class *software language*. Such investigations may also lead to refined competencies as an answer to the question: *What is a software language entity used for?*

4.3.3 Indicator Discovery

RQ W2.2: What features indicate articles' relevance to the domain class?

For imbalanced datasets, a classifier usually overfits towards the major class unless countermeasures are taken [Chawla et al., 2002]. Our training data contains a random sample. The random sample is representative for the distribution in the whole set of articles.

Adding the seed articles to the training data has two observable effects. i.) The distribution with regard to an article's relevance in the training data is not representative anymore as we added only additional software language articles. This can be seen as a form of oversampling. Using synthetic oversampling with SMOTE further improves results. ii.) The feature selection based on *Gini index* or *entropy* pick features according to how well they recall that an article is on a software language or not. The features considered by the learned decision tree are presented in Figure 4.12. In the scope, the Non-software language articles are very diverse and cover different domains. Hence, the features that recall the negative class have a lower score than the features that recall the positive class. Consequently, the features that appear in software language articles are favored by the feature selection. Therefore, *the machine learning algorithm isolates articles on software languages from articles on many other domains.* The effects can be observed in the learned decision tree. The plotted decision tree reveals discrimination scores for each feature and how many articles arrive during the training phase in which branch[38]).

Infobox Templates: *file format, programming language;* **URL Pattern:** *programming, language;* **Lemmas:** *syntax, code, programming, compiler, design, general-purpose, language, support, compile, object-oriented, use;* **Hypernyms:** *language;* **Wikipedia List Entries:** *List of programming languages, List of programming languages by type, List of file formats, List of C-family programming languages, List of object-oriented programming languages.*

Figure 4.12 – The list of features used by the decision tree, where features positively recalling a software language are more frequent than others.

Promising indicators give away what makes a software language recognizable on Wikipedia. In Figure 4.12, we list the features used by the learned decision tree, in which a feature can appear multiple times. The infobox templates 'programming language' and 'file format' provide a strong positive indication, but they do not exist for every relevant article. While the hypernym 'language' also provides strong indication, it cannot be used alone. Otherwise, natural language related articles are confused to be relevant. Lemmas, lists and URL pattern help in discriminating relevant articles from articles with overlapping features.

[38] `https://github.com/softlang/wikionto/blob/master/data/datasets/sltreeKBest_23_mutual_info_classif_Oversampling_name.pdf` — Requested March 31, 2022

4.3.4 Article and Category Relevance

RQ W2.3: How many articles and categories remain relevant for the domain class?

Table 4.3 summarizes the degree of the reduction per root category based on the configuration from Section 4.3.1. That is, we significantly augment the identification of software languages. Based on the predicted reduction, we find 2797 (∼2.68%) more articles in the scope of two root categories than there are already in the seed, which only contains 327 (∼0.31%) articles.

For the root category 'Computer file formats', the classifier predicts that only about 12% of the articles are relevant. When inspecting infobox templates used in these articles, we observe a topic mix with, for example, software, websites and companies. Here, a threat remains that the articles on formats are just not recognizable, because the format by itself is not *notable*. In comparison to 'Formal languages', a higher percentage of categories is estimated as relevant. Accordingly, we observe that the subcategory tree is more consistently maintained.

	Formal languages	**Computer File Formats**
Total Articles	101641	6116
Seed " (%)	301 (∼0.30%)	46 (∼0.75%)
Relevant " (%)	2897 (∼2.85%)	745 (∼12.18%)
Total Categories	21822	235
Seed " (%)	253 (∼1.16%)	18 (∼7.66%)
Relevant " (%)	1339 (∼6.14%)	79 (∼33.62%)

Table 4.3 – How many articles and categories exist in *total*, identified by the *seed*, and classified as *relevant*.

Inspecting single categories backs up subjective hypotheses based on manual inspection of the usefulness of specific categories. For example, out of 22546 articles that are (transitive) members 'Statistical data types' only 52 are classified as relevant. The number of articles indicated to be relevant in a category becomes a more objective estimation for its usefulness.

Table 4.3 provides a summary of inspecting all categories based on the assumption that the amount of relevant articles hints at the relevance of categories. Out of 21822 categories below 'Formal languages', 353 categories contain seed members and 1339 categories remain relevant. Only 6% of transitive subcategories under 'Formal languages' are estimated to be relevant within the scope.

4.3.5 Software Language Classification

RQ W2.4: Which classification dimensions for software languages can be identified on Wikipedia and aligned with literature?

We aim to identify classification dimensions for software languages as initially discussed in Section 2.3.1. Here, we consider cherry picked papers covering a broad range of languages, such as [Shilov et al., 2012], instead of papers inspecting a small subset as in [Sarimbekov et al., 2016]. For each classification dimension, we manually identify a literature reference and an exemplifying category that is linked to relevant articles. Table 4.4 serves as a summary of our initial effort on extracting properties from discovered articles.

Classification dimensions can be extracted from Wikipedia's category graph [Nastase and Strube, 2008, Lämmel et al., 2013a]. For software languages, aside from categories that cover historical information, which is also described in literature [Sammet, 1972] or in

Dimension	*Literature*	*Category*
Historical	[Sammet, 1972]	Programming languages created in the 1990s
		Programming language families
Paradigms	[Floyd, 1979]	Programming paradigms
Abstraction Level	[Kahanwal, 2013]	Low-level programming languages
		High-level programming languages
Naturalness	[Shilov et al., 2012]	Non-English-based programming languages
Purpose	[Shilov et al., 2012]	Educational programming languages
		Bibliography file formats
Domain-specific	[Fowler, 2010]	Domain-specific programming languages
Pragmatics	[Shilov et al., 2012]	JVM programming languages

Table 4.4 – Recognized classification dimensions from literature in terms of exemplary Wikipedia categories.

a well-known poster from O'Reilly[39], paradigms [Floyd, 1979] and purpose [Shilov et al., 2012, Anureev et al., 2008] are frequent dimensions. In literature, there are many more fine-grained dimensions: The level of abstraction [Kahanwal, 2013]; fine-grained classification of syntax, e.g., by its naturalness, degree of semantics formalization and pragmatics [Shilov et al., 2012]; and whether it is domain-specific [Fowler, 2010].

[39] `https://www.cs.toronto.edu/~gpenn/csc324/PLhistory.pdf` — Requested March 31, 2022

4.4 Conclusion

In this chapter, we have presented a methodology to isolate a corpus of articles that are relevant to a single ontological class. Such articles help with understanding the instances, e.g., Java or MATLAB in the case study, and the class itself, in our case software language.

In the case study, only a small set of 327 relevant Wikipedia articles can be identified in advance and are then used as a seed. We identify strong indicators for discovering 2797 more articles on software languages. By focusing on indicators, the analysis results also help us to understand what structured information in an article is central and should actually be common in all relevant articles, e.g., an infobox using a relevant template or a precise hypernym in the first sentence. This way, the indicators may also help with authoring idiomatic software language articles in the future. Additionally, the features can be used to systematically recover different types of properties for a software language and then identify similar instances.

We show that a learned decision tree classifier provides reasonably high recall and low false-positive-rate and allows one to inspect isolated articles inside Wikipedia. While learned random forests might provide higher accuracy, we are specifically interested in higher interpretability of decision trees. 31 domain experts evaluated our classifier by labeling over 990 Wikipedia articles in a survey. In the survey, we encountered similar disagreements between experts as identified in [Combemale et al., 2017] and hence emphasize the need for further efforts to investigate on the borders of the term *software language* as an ontological class with a focus on its competencies.

Chapter 5

Mining Technology Usage

A reoccurring problem in software engineering is how to share knowledge and experience on the usage of a software technology. Official documentation typically discusses crafted examples instead of complex software. With the rise of open source software, software engineers have the opportunity to inspect complex software, for which GitHub is the most prominent platform. In this chapter, we aim for an approach on how to query and report patterns of technology usage on GitHub.

The key pillar of our methodology is the declarative rule-based query engine QegaL. A technology usage pattern is informally defined and encoded in terms of a set of inference rules. Then, a query engine is used to incrementally extract facts from selected repositories. Extracted facts including links to code that exemplifies the technology usage are persisted in triple stores. We perform a case study in which we develop an emerging catalog of patterns of EMF usage. First, we identify 1438 repositories using EMF on GitHub. Then, we detect and report on the frequency of usage for ten common EMF usage patterns.

Chapter contribution: This chapter summarizes research on exploring EMF usage on GitHub [Härtel et al., 2018b]. The general methodology to extract facts from GitHub is the contribution by Johannes Härtel. The case study published in [Härtel et al., 2018b] is a contribution of this thesis.

5.1 Mining Technology Usage on GitHub

In this Section, we summarize the methodology on mining patterns of technology usage from [Härtel et al., 2018b] that was contributed by Johannes Härtel. The research is dedicated to the following research question:

RQ T1: How can we locate traces of technology usage on GitHub?

The methodology follows a linear step sequence: *i.) Define a pattern of technology usage, ii.) Locate repositories using the technology, iii.) Select repositories, iv.) Develop detection of pattern instances, v.) Report results.* We summarize the steps as follows.

We aim to better understand how a technology is used in GitHub projects. To this end, we develop an emerging catalog of *technology usage patterns*. In the case study on EMF in Section 5.2, we define patterns to investigate on consistency and completeness of interrelated artifacts. For example, we search for a metamodel, a code generation configuration that refers to the metamodel, and generated Java code. The patterns are described in a textual informal manner.

We then *locate GitHub repositories* that potentially use the patterns. Therefore, we execute a query with GitHub's search engine to identify candidates based on file extensions or code fragments that are as unique as possible for the technology, e.g., we search for files with a '.java' file extension and the code fragment 'extends EObject '.

As the next step, we *select repositories* that fulfill criteria so that analysis is not distorted by particularities of individual projects. In the case study, we focus on vanilla EMF repositories, which contain a Manifest file, which is EMF's default for defining OSGi projects and dependencies.

For each pattern, we develop algorithm(s) to *detect it* in a repository, which includes analysis of program as well as model artifacts. Here, we propose the use of the rule-based engine QegaL that is described in Section 5.1.1.

At last, we *report the results* from the detection by counting identified instances and informal interpretation of retrieved numbers. At this point, looking into a sample of repositories and the empirical findings can lead to more detailed insights.

5.1.1 Detection of Pattern Instances with QegaL

Mining software repositories is a common activity in software engineering with diverse use cases such as understanding project quality [Tufano et al., 2017], technology usage [Rocco et al., 2018, Rocco et al., 2020], and developer profiles [Hauff and Gousios, 2015]. Such mining activities involve, more often than not, a phase for data extraction from the source code in the repository with recurring tasks such as processing the folder structure (possibly in the commit history), classifying repository artifacts (e.g., in terms of the languages or technologies used), and extracting facts from the artifacts by parsing or otherwise.

The declarative rule-based language QegaL is the key pillar of our approach. It supports uniform, inference-based extraction of facts from a repository (the file system); the artifacts in the repository (their content) are referenced by URIs; previously extracted facts are further processed by reasoning; all inferred facts are maintained in a triple store. In our implementation, we use Apache Jena for which we provide dedicated language support tailored towards mining software repositories. To give the reader an idea, consider the following trivial rule drawn from the case study in Listing 5.1:

```
1 (?x sl:manifestsAs sl:File)
2 (?x sl:elementOf sl:XML)
3 Extension(?x,"ecore") →
4  (?x sl:elementOf sl:Ecore).
```

Listing 5.1 – Sample rule classifying Ecore files.

The body of the rule (i.e., the premise before '→') quantifies over artifacts *?x* that are files (line 1) with extension 'ecore' (line 2) and readily known to be members of *XML* (line 3). For every matched artifact, the head of the rule (i.e., the conclusion right to '→') states that the artifact is also a member of *Ecore*. This way, the rule infers triples to detect instances of an *Ecore* model.

QegaL relies on *primitives* for accessing the repository (folder structure and content of files). In particular, the body of a rule may use such primitives to match or bind repository data such as file names and file content in terms of different representations (e.g., ASTs). Primitives may be also used in heads of rules for the purpose of data manipulation or for expanding given bindings into sets of inferred triples.

The primitives needed in the case study are presented in Table 5.1 with some omissions for brevity. They are free of side effects to the repository. A primitive takes one or more arguments. Each argument is either a URI, a literal, or a placeholder to be bound. The application of a primitive may fail, if placeholders cannot be bound to valid results or the given arguments are not in the relationship expressed by the rule.

The rule-based approach is declarative and modular, when compared to the common use of problem-specific custom functionality for processing folder structure and file content (e.g., [Karus and Gall, 2011, Rocco et al., 2018]). It leverages an extensible suite of accessor primitives for interacting with standardized formats and structures such as *Java*, *XML*, and the file system (the folder structure) in a uniform manner. The rule-based approach helps in manifesting only the facts that are actually needed, as opposed to operating on complete ASTs or similar structures (as in, e.g., [Roover, 2011, Roover et al., 2013, Atzeni and Atzori, 2017, Shatnawi et al., 2017]); it is up to the rules and the accessor primitives to selectively extract and infer more facts. Such inference is similar to the event-condition-action paradigm [Dittrich et al., 1995].

Primitive	Parameters and Description
IsFile	(artifact) Matches if the artifact URI can be accessed as a file.
IsFolder	(artifact) Matches if the artifact URI can be accessed as a folder.
Extension	(file, extension) Matches if the file URI has a given extension. (When extension is a placeholder, it will be bound.)
Children	(uri, $part_1$, ..., $part_n$) Decompose a uri into its parts split at '/'; parts filled in are matched; variable parts are bound.
DecFs	(folder, xpath, result) Decompose references to file system by applying an XPath expression xpath on the repository starting from the given folder; assigns the URI of the first result to result.
DecFs	(folder, subj, pred, obj) A variation on the previous primitive. It infers a *set* of triples, when used in the head of a rule. The inferred triples vary in the subject based on the argument subj which corresponds to the XPath expression in the basic form of DecFs. The arguments pred and obj are assigned to regular URIs. For instance, DecFs(repository:/,"/*", sl:partOf, repository:/) adds a sl:partOf triple for all first-level repository children (XPath '/*' for subject) of the repository (for the object).
XmlWellformed	(file) Parses the content referred to by the file URI and matches if the content is well-formed *XML*.
DecJava, DecXml	Variations on DecFs working on *Java* ASTs or *XML* trees as opposed to the file system.
StrXml, StrJava, UriXml, UriJava	Variations on the decomposition primitives for use in rule bodies, as described above. These variations perform data lookup as opposed to constructing a reference. The Str... primitives look up a string (e.g., an attribute in XML) and return it as a result. Likewise, the Uri... primitives look up a string which is a URI.

Table 5.1 – Primitives for accessing folder structure (in general) and file content for *XML* and *Java* (as needed in the case study).

5.2 A Case Study on EMF Patterns of Usage

Next, we design and execute the case study for analyzing EMF usage in projects on GitHub. In this large-scale case study, we examine 5759 repositories. Here, we limit ourselves to only studying the most recent version of each project, leaving an evolutionary analysis out of scope. We locate projects with traces of EMF usage on GitHub. Then, we assess these projects in terms of some basic architectural characteristics to prepare a selection of well-understood project layouts for which a mining process is assumed to provide more comprehensible insights. Eventually, we detected EMF patterns of usage as introduced in Section 5.2.1. We describe these phases in the following subsections and summarize our findings. The artifacts as well as step by step guide on how to reproduce case study results are available online[1].

5.2.1 Define Patterns of EMF Usage

EMF can be used in different ways in projects, subject to the *presence* of different types of artifacts, possibly with different *multiplicities* and in different *combinations*. Here, we develop an emerging catalog for EMF which covers these basic 'artifact' types: *Ecore Package*, as identified in '.ecore' files where one such file can possibly define several such packages; *Java Package* – an actual Java package containing derived classes according to a metamodel, a factory, and a package description; and *Generator Package* as identified in '.genmodel' files.

Artifacts of these types are related in traceable ways in a project. In fact, by querying for types of artifacts and relations, e.g., by determining the *absence* of specific types of artifacts or the *absence* of an expected relationship, we may infer cases of *incompleteness* or *inconsistency*, where these are either *potential* or *definite* problems of usage or, in fact, of maintaining EMF usage in the repository. Table 5.2 lists patterns organized along these different dimensions (artifact type, presence, incompleteness, inconsistency, potential versus definite). We group by cardinalities of artifacts: single, double, and triple artifact patterns. For brevity, we exclude patterns related to XMI-based persistence in this paper.

5.2.2 Locate EMF Repositories

We used the GitHub search API to locate all recently indexed *Ecore*, *Generator Model* and *Java Model* files on GitHub as an indication of EMF usage in repositories. The corresponding queries are listed in Table 5.3. For what it matters, the API search limit is circumvented by recursive query segmentation which splits a query by setting an upper and lower bound in file size based on the returned number of total results. This process may miss some results. The search API only considers heads of default branches and files smaller than 384 KB. A list of 5759 GitHub repositories was extracted from the query results.

5.2.3 Select Vanilla EMF Repositories

We used queries to initially recover the repository layout in terms of used build systems, project dependencies, and other aspects. We developed the following classifiers for repositories as an extension to the pattern catalog in Table 5.2.

- *Homogeneous* versus *heterogeneous build system:* We search for traces of *Manifest*, *POM*, *Gradle*, and *ANT*. The rules work in analogy to what is shown for an Ecore model. Hence, they focus on a file with a specific file extension, e.g., 'pom.xml' for *POM*. In the homogeneous case, only one such technology is detected; otherwise a repository has heterogeneous build systems. We assume that the heterogeneous situation is harder to understand in terms of project dependencies.

[1] `https://github.com/softlang/qegal` — Requested March 31, 2022

Id	Cls.	Artifacts	Description and cause
Single artifact patterns			
E	Pres.	• Ecore Pkg.	The presence of an Ecore Pkg. in '.ecore' files as root or subpackage.
J	Pres.	• Java Pkg.	The presence of a Java Pkg.
G	Pres.	• Genmodel Pkg.	The presence of a Genmodel Pkg. in '.genmodel' files as root or subpackage.
C	Pres.	• Customized Java Pkg.	The presence of a Java Pkg. with customized interface or implementation.
Double artifact patterns			
EJ1	Pot. Incomp.	• Ecore Pkg. • Java Pkg. (m[a]) [a]Missing	A Java Pkg. cannot be found for a given nsURI as extracted from some Ecore Pkg. This is only a potential incompleteness, because a Java Pkg. could be potentially derived, if no customization is intended.
EJ2	Def. Incomp.	• Ecore Pkg. (m) • Java Pkg.	An Ecore Pkg. cannot be found for a given nsURI as extracted from some Java Pkg. This is a definite incompleteness because the Java Pkg. is derived and thus, the underlying primary artifact (the Ecore Pkg.) should also be in the repository.
EJ3	Pres.	• Ecore Pkg. • Java Pkg.	The presence of a Java Pkg. and Ecore Pkg. with the same nsURI. One Ecore Pkg. can correspond to many Java Packages.
EE	Def. Incons.	• Ecore Pkg. • Ecore Pkg.	An Ecore Pkg. with at least one competing Pkg. with the same nsURI.
EJc1	Def. Incons.	• Ecore Pkg. • Java Pkg.	A Java class that is part of the Java Pkg. with a corresponding Ecore Pkg., but without a corresponding Ecore classifier (based on name comparison). For instance, one may have forgotten to remove a Java class derived from an earlier version of the metamodel.
EJc2	Def. Incons.	• Ecore Pkg. • Java Pkg.	An Ecore classifier contained in an Ecore Pkg. with a corresponding Java Pkg., but without a corresponding Java classifier (based on name comparison). The Java Pkg. is thus out of sync with the Ecore Pkg. in the repository.
Triple artifact patterns			
EJJ	Pres.	• Ecore Pkg. • Java Pkg. • Java Pkg.	An Ecore Pkg. with at least two corresponding Java Packages.
EJG	Pot. Incomp.	• Ecore Pkg. • Java Pkg. • Generator Pkg. (m)	For a corresponding pair of Java Pkg. and Ecore Pkg., a corresponding Generator Pkg. cannot be found.

Table 5.2 – An EMF Repository Pattern Catalog A selection of the detection rules is discussed in Section 5.2. Every rule is available online.

Evidence	Query	Extension
Java Model	"extends EObject {"	java
Ecore Model	GenModel	ecore
Generator Model	EClass	genmodel

Table 5.3 – Queries for locating repositories through GitHub API.

- *Single component* versus *multiple components:* Based on an analysis of project dependencies, we determine the number of components. We assume that repositories with multiple components are special. Such a repository may be, for example, a 'zoo' [Kusel et al., 2013]. Note that a single component can still imply the presence of multiple (dependent) projects.

- *Variants:* This classifier applies when we locate different versions of the same project in a repository based on the analysis of project dependencies. We assume again that repositories with variants are special. Such a repository may capture, for example, versions in a migration process. We detect duplicates by checking uniqueness of the

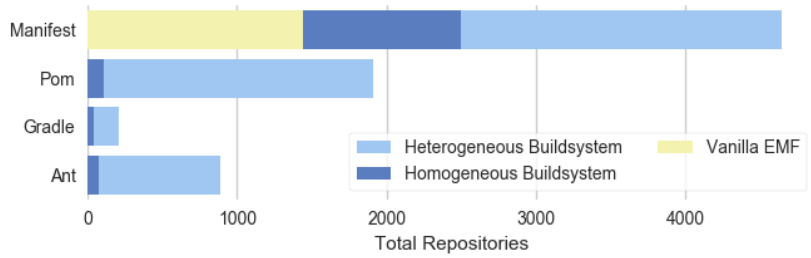

Figure 5.1 – Number of repositories with a particular build system further partitioned into homogeneous versus heterogeneous case.

value for "Bundle-SymbolicName" in manifest files.

We use additional QegaL rules for assessing basic architectural characteristics of the located projects. By default, Manifest files are used in EMF projects for declaring OSGi projects and dependencies. We decided to only include homogeneous repositories using Manifest files for mining. The analysis of dependencies is based on 'Bundle-SymbolicName' elements in Manifest files. Listing 5.2 presents the rules for inferring the occurrence of declarations (predicate *sl:decOccurs*) and references (predicate *sl:refOccurs*) and OSGi dependencies between Manifest files (predicate *sl:dependsOn*):

```
// Extraction of Bundle-SymbolicName declaration.
(?file, sl:elementOf, sl:Manifest)
StrManifest(?file, "Bundle-SymbolicName", ?x)
// Replace all details.
ReplaceAllToUri(?x, "(;[^,]*)|\\s", "", ?declaration) →
(?file, sl:decOccurs, ?declaration).

// Extraction of Bundle-SymbolicName references.
(?file, sl:elementOf, sl:Manifest)
StrManifest(?file, "Require-Bundle", ?x)
// Replace all strings.
ReplaceAll(?x, '("[^"]*)"', "", ?xi)
// Replace all details.
ReplaceAll(?xi, "(;[^,]*)|\\s", "", ?references) →
SplitToUri(?references, ?file, sl:refOccurs, ',').

// Creating dependency structure
(?a, sl:elementOf, sl:Manifest)
(?b, sl:elementOf, sl:Manifest)
(?a, sl:decOccurs, ?deca)
(?a, sl:refOccurs, ?decb)
(?b, sl:decOccurs, ?decb) →
(?deca, sl:dependsOn, ?decb).
```

Listing 5.2 – Rules for extracting OSGi declarations, references and dependencies.

The primitive `StrManifest(file, property, value)` is a specialized decomposition of a Manifest file, comparable to `StrJava` in Table 5.1; it binds *value* to a Manifest property in literal form. In the rules shown above, it binds *?x* to the required or defined bundles in string representation. The chains of `ReplaceAllToUri`, `ReplaceAll` and `SplitToUri` process *?x* in that it can be added to the model as declaration or reference. We exclude repositories with duplicated declarations (*sl:decOccurs*) for the same URI (classifier *Variants*). Then, we apply an algorithm for detecting connected components to the *sl:dependsOn* triples, as inferred by the last rule shown above. At last, we exclude repositories with multiple components.

The results of the selection steps are depicted in Fig. 5.1. We only consider repositories with a single component, Manifest usage only, and no variants. We refer to these repositories

as 'Vanilla EMF repositories'. There are 1438 such projects. These are the projects considered for detecting pattern instances.

5.2.4 Detect EMF Patterns of Usage

To exemplify how code queries are developed, we discuss the detection of correspondence between Java and Ecore models; we show the decomposition of an Ecore model; we omit the rules for *Java* model detection in the set of available *Java* files; we also omit handling Ecore sub-packages. Consider the beginning of a small Ecore sample file in Listing 5.3.

```
<ecore:EPackage ... name="fsml"
nsURI="http://www.softlang.org/metalib/emf/Fsml"
nsPrefix="fsml">
<eClassifiers xsi:type="ecore:EClass" name="FSM">
...
```

Listing 5.3 – The first lines of a sample Ecore file.

Any Ecore model provides two kinds of fragments that we are primarily interested in. Those are package elements (see line 1-3 in Listing 5.3) and classifiers (see line 4 in Listing 5.3). Hence, the rules in Listing 5.4 decompose an Ecore model into root package and nested classifiers.

```
// Detect the root package.
(?x, sl:elementOf, sl:Ecore) →
DecXml(?x, "/ecore:EPackage", sl:partOf,?x)
DecXml(?x, "/ecore:EPackage", sl:elementOf, sl:EcorePackageXMI).

// Detect the nested class in a package.
(?x, sl:elementOf, sl:EcorePackageXMI) →
DecXml(?x, "/eClassifiers", sl:partOf, ?x)
DecXml(?x, "/eClassifiers", sl:elementOf, sl:EcoreClassifierXMI).
```

Listing 5.4 – Decomposing an Ecore model into classifiers.

Next, the rules in Listing 5.5 assign the *nsURI* to the root package and the *URI* for each classifier. As a result, a *sl:partOf* relationship is inserted along the nesting structure and fragments are classified by *sl:EcorePackageXMI* and *sl:EcoreClassifierXMI* respectively. This decomposition is handled by the first two rules using the `DecXml` to construct URIs in the repository referencing scheme. In contrast, the last two rules extract the attributes 'nsURI' and 'name' using `UriXml` and `StrXml`. The primitives return the recovered content directly as string or URI, as we need the actual attribute values 'FSM' (name) and '`http://www.softlang.org/metalib/emf/Fsml`' (nsURI) as exemplified in Listing 5.3.

```
// Recover NsUri for a package as URI.
(?x, sl:elementOf, sl:EcorePackageXMI) →
UriXml(?x, ?x, sl:nsUri, "/@nsURI").

// Recover uri for a class
(?classifier, sl:elementOf, sl:EcoreClassifierXMI)
// Get package's nsURI.
(?classifier, sl:partOf, ?package)
(?package, sl:nsUri, ?nsUri)
// Get the class' name as string.
StrXml(?classifier, "/@name", ?classifierName)
// Build a compound ?uri.
UriConcat(?nsUri, '#//', ?classifierName, ?uri) →
// Add Uri for a classifier, i.e., nsURI with appended name.
(?classifier, sl:uri, ?uri).
```

Listing 5.5 – Decomposing Ecore into classifiers appending a nsURI.

A correspondence relationship between Ecore packages, i.e., elements of *sl:EcorePackageXMI*, and Java packages, i.e., elements of *sl:EcorePackageJava*, is established by matching the nsURI. The rules are presented in Listing 5.6.

```
(?xmi, sl:elementOf, sl:EcorePackageXMI)
//Recover nsURI assigned to package in .ecore
(?xmi, sl:nsUri, ?nsUri)
(?java, sl:elementOf, sl:EcorePackageJava)
//nsURI exactly matches the nsURI for a Java Package
(?java, sl:nsUri, ?nsUri) →
(?xmi, sl:correspondsTo, ?java).
```

Listing 5.6 – Rules recovering the correspondence between Ecore- and Java packages.

In analogy, the rules in Listing 5.7 add the correspondence relationship between the Ecore- and Java classes based on the *URI*. The *Java* model extraction is based on accessing the nsURI property in the *Java* AST by a similar primitive `UriJava` (not shown here). Correspondence relationship between classes is established by matching the nsURI concatenated with the classifier name.

```
(?xmiClassifier, sl:elementOf, sl:EcoreClassifierXMI)
//Recover uri assigned to classifier in .ecore
(?xmiClassifier, sl:uri, ?uri)
(?javaClassifier, sl:elementOf, sl:EcoreClassifierJava)
//URI exactly matches the uri for a Java class
(?javaClassifier, sl:uri, ?uri) →
(?xmiClassifier, sl:correspondsTo, ?javaClassifier).
```

Listing 5.7 – Rules recovering the correspondence between Ecore- and Java packages.

5.2.5 Report Results

The results of the case study for 1438 Vanilla EMF repositories are shown in Figure 5.2, Figure 5.4, and Figure 5.3. The discussion does not aim to be comprehensive. Overall, the online corpus features additional results that can be reproduced. At the most basic level, we show numbers of repositories per pattern and numbers of pattern instances (Figure 5.2).

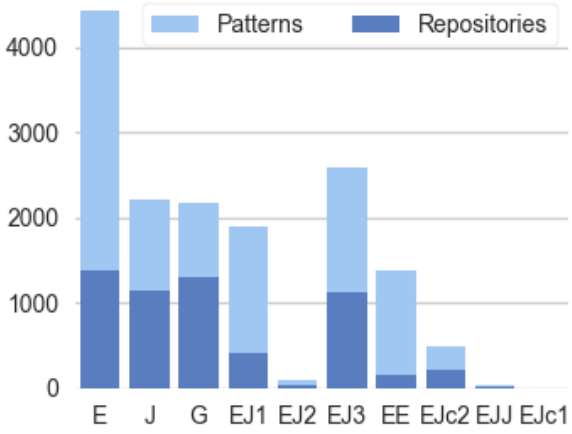

Figure 5.2 – Overall number of detected pattern instances and the number of repositories, where a pattern is found.

The median pattern occurrence of Ecore (E), Java (J) and Genmodel (G) packages and the regular correspondence (EJ3) for a single repository is 1 (Figure 5.4). This indicates that common usage is concerned with only one package. The coefficient of variation for measuring the relative variability 'cv' is the highest for EJ2, EE and EJJ — the first two patterns indicate problems; the latter pattern represents a very rare case (1% of the Vanilla repositories,

where one Ecore package is matched with two Java Packages). We expect corner cases and problems to have high variations in pattern occurrence. Having no package correspondence (EJ1) or a regular package correspondence (EJ3) can both be considered as common usage. Forgetting to remove Java classes (EJc2) can also be considered a common usage despite being a definite inconsistency. On the contrary, we detected no repository missing a Java classifier for an Ecore classifier (EJc1).

	E	*J*	*G*	*EJ1*	*EJ2*	*EJ3*	*EE*	*EJc2*	*EJJ*
Sum	4427	2217	2181	1894	96	2598	1376	496	45
Repo	1389	1152	1294	404	43	1127	157	223	18
mean	3.1	1.5	1.5	1.3	0.1	1.8	1.0	0.3	0.0
std	9.8	3.2	2.2	6.2	0.5	7.3	8.3	1.6	0.5
25%	1.0	1.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0
50%	1.0	1.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0
75%	2.0	1.0	1.0	1.0	0.0	1.0	0.0	0.0	0.0
max	261	71	28	103	10	248	251	26	16
cv	3.2	2.1	1.5	4.7	8.2	4.0	8.6	4.5	15.2

Table 5.4 – The distribution of detecting a pattern in the repositories (where the minimum is always 0.0). Row 'cv' lists the coefficient of variation.

In Figure 5.3, we examine the projects on scales for different characteristics (forks, stars, and mining time) to see how the size of projects relates to these characteristics and also how the relative frequency of pattern-based problems or the absence thereof relates to these characteristics. For instance, we can observe (right-bottom and right-middle charts in Figure 5.3) that definite incompleteness and inconsistency is of much less or no concern with increasingly more forked or stared repositories.

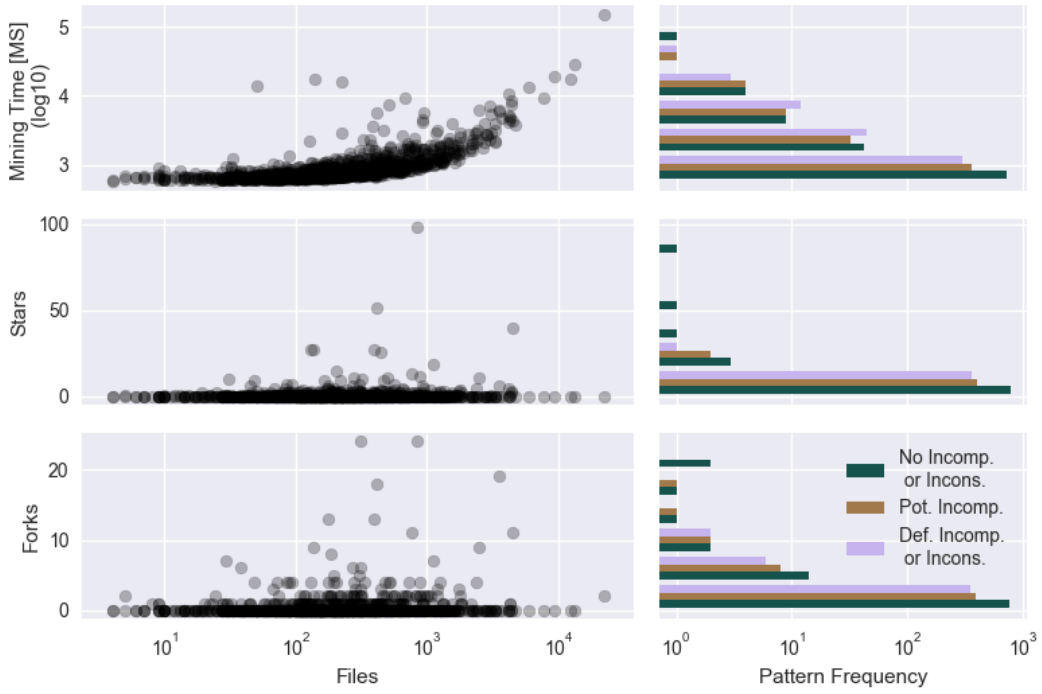

Figure 5.3 – Different repository and mining characteristics shown with respect to the amount of files in a repository. The right column shows how often different types of patterns occur for some histogram buckets and for the different characteristics.

5.2.6 Threats to Validity

We briefly discuss the threats to validity for the case study on mining EMF patterns of usage. The location and selection of the repository list towards 'Vanilla EMF' can be seen as a threat to external validity. We might miss 'important' repositories and thereby produce biased results. Due to the complexity of EMF and the diversity of possible repository layouts, we potentially miss particular pattern instances. We extensively tested our rules in an instance- and parameter-based manner to cover this internal threat, but some rules are incomplete (e.g., considering only one type of build system) or approximates (e.g., in assuming a very strict naming scheme for generated classifiers). Our pattern catalog and the rating of patterns, i.e., *potential* or *definite incompleteness* or *inconsistency*, are potentially subjective, even though we have extensive experience with EMF and manually debugged when developing the inference rules.

5.3 Conclusion

In Section 5.2.5, we have provided insights into empirical investigations on the basic usage of EMF on GitHub, where we have covered variation points, potential incompleteness and consistency problems. This way, we have contributed towards detecting technology usage in repositories based on a modular rule-based approach.

The case study shows that at least 5759 repositories using EMF exist on GitHub. There may still be more repositories that are not vanilla EMF repositories and need to be identified by other queries. The results of queries for selecting 1438 vanilla EMF repositories also show that nearly 2000 repositories exist which use Maven for dependency management instead of Manifest files, which leaves room for discussion and further analyses that are tailored towards the projects that are setup up with Maven.

Chapter 6

Modeling Technology Usage

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 stakeholders consult scattered resources, where *textual explanations* and *code examples* are provided. In this chapter, we propose the use of interconnected megamodels that serve as macroscopic summaries of technology usage. More specifically, we provide a methodology for their construction that focuses on reproducibility and robustness against misconception. We interconnect model elements with discovered textual explanations and code examples. The discovery relies on the systematic reduction of a corpus to evidence for each megamodel increment. The manual effort is based on systematically debugging dedicated queries. We perform a case study, where we apply the methodology for a technology model that summarizes EMF code generation. EMF itself is complex enough so that research is dedicated towards providing a better understanding, e.g., by identifying common problems mentioned in Eclipse forums [Kahani et al., 2016]. Then again, the code generation scenario is simple enough as a common set of files can be identified in the layout of a project using EMF. We perform a second case study, where we emphasize the methodology's robustness and present how it can be used to reveal a misconception for EMF usage.

Chapter contribution: This chapter presents research on creating megamodels of technology usage in a reproducible manner as published in [Heinz et al., 2020]. We contribute a methodology for the reproducible construction of technology models and two case studies. In one of the case studies, the technical mining framework as described in Chapter 5 is used again.

6.1 Technology Models

In this chapter, we are concerned with the aspect that, within the context of education or for the purpose of a useful technical documentation of a software technology, any technology model needs to be constructed in a reproducible manner so that it can be safely reused and referred to. To address this concern, we develop a methodology in Section 6.2 and perform case studies on EMF in Section 6.3 and Section 6.4. We aim to answer the following methodological research question by means of the developed methodology and two case studies.

> *RQ T2: How can we construct a technology model in a reproducible manner so that it is interconnected with existing textual explanations and code examples?*

In non-scientific literature, we observe that EMF code generation is explained and modeled in terms of loose visual diagrams, for example, in text-based tutorials.[1] In MDE literature, many researchers have summarized code generation using EMF in different contexts such as developer activities [Hebig et al., 2012], adapted EMF processes [Ed-Douibi et al., 2016] and pluggable analysis [Härtel et al., 2017]. EMF has been used as an exemplary technology in case studies several times in our research [Favre et al., 2012c, Heinz et al., 2017, Härtel et al., 2017, Schauss et al., 2017, Härtel et al., 2018b].

The technology model that is depicted in Figure 6.1 serves as the running example for the first case study in Section 6.3, where we aim to get the reader acquainted with executing the methodology in detail. It provides a visual summary of the central artifact types and their relations that are often covered in developer literature as well as scientific literature. 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 [Heinz et al., 2017, Härtel et al., 2017, Zaytsev, 2012, Favre et al., 2012c, Lämmel and Varanovich, 2014].

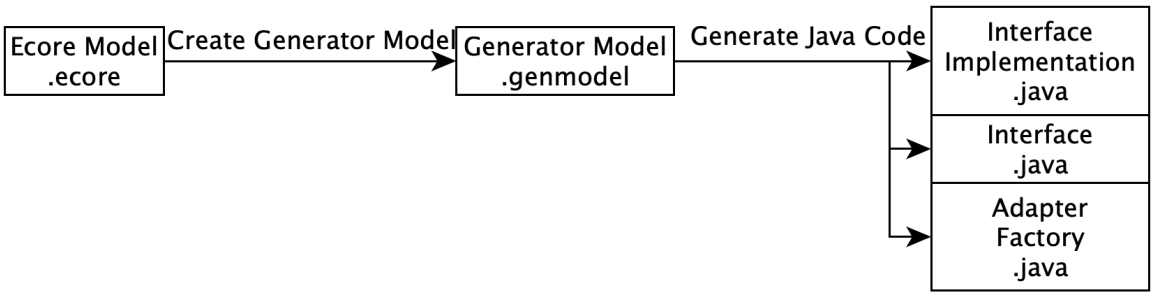

Figure 6.1 – A technology model of EMF code generation.

The scope of a technology model is not only concerned with API usage. API usage research focuses on discovering pattern in how a technology's API is used on the level of classes and methods [Robillard et al., 2013]. A technology model includes artifacts beyond programs such as configuration or data models and focuses on their relationships, e.g., a configuration file can be in a consistency relationship with an input data model and defines the mapping to generated code. The model presented in Figure 6.1 is the starting point for a library of technology models that are dedicated to share knowledge on EMF usage. The artifacts produced for the case studies are available online.[2]

[1] `https://eclipsesource.com/blogs/tutorials/emf-tutorial/` — Requested March 31, 2022
[2] `https://github.com/softlang/megaemf` — Requested March 31, 2022

6.2 Reproducible Model Construction

We develop an incremental process to construct technology models. Technology models consist of technology-specific artifact types and their relations. In the reproducible process, artifact types and relations are added as increments one after another. For every increment to the model, evidence is needed. Hence, textual explanations and code examples need to be identified and linked. Linked textual explanations and code examples add value to the "meaningless diagram".

6.2.1 Textual Explanations & Code Examples

Textual explanations and code examples have been identified as essential resources for understanding API (method) usage [Robillard, 2009, Roover et al., 2013, Robillard and Chhetri, 2015]. We assume that the problem of identifying such resources naturally transfers to understanding technology usage beyond APIs. 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 common technology usage. Tutorials and forum posts can be consolidated, but their structure and quality may challenge a systematic exploitation. ii.) *Scientific literature* tends to provide more abstract *textual explanations*. iii.) *Demo projects* are typically linked in *developer literature* to provide a collection of referential *code examples* that represent suggested common usage. iv.) *Wild projects*, such as open source projects on GitHub, provide more complex *code examples* that represent actual technology usage. The diversity of technology usage has been explored in [Kolovos et al., 2015]; potential inconsistencies of usage have been analyzed in [Härtel et al., 2018b].

6.2.2 Query-based Reproducibility

By documenting where and how the evidence is identified, the construction process becomes reproducible. For code examples, this task can be largely automated in analogy to detecting instances of pattern of technology usage as described in Chapter 5. Searching for textual explanations in a corpus without sophisticated natural language processing approaches requires manual effort. For code and text, manual effort is needed when relevance and quality need to be assured based on expertise.

Figure 6.2 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 can be linked and documented immediately. Otherwise, a promising *corpus* that contains evidence needs to be selected. As long as *evidence* is missing, the goal is to reduce the corpus by executing several manual as well as automated steps. 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 patterns that are identified in previously *linked evidence* and returned *query results*. The *query results* are then manually inspected to confirm and link concise explanations and idiomatic code examples. 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 types of corpora are processed. They can be even processed in an interleaving manner.

The first case study in Section 6.3 gets the reader acquainted with executing the methodology in detail. The second case study in Section 6.4 discusses the methodology's robustness. It demonstrates how executing the methodology helps to reveal misconceptions in a technology model. In the second construction process, a relation is added to the technology model which

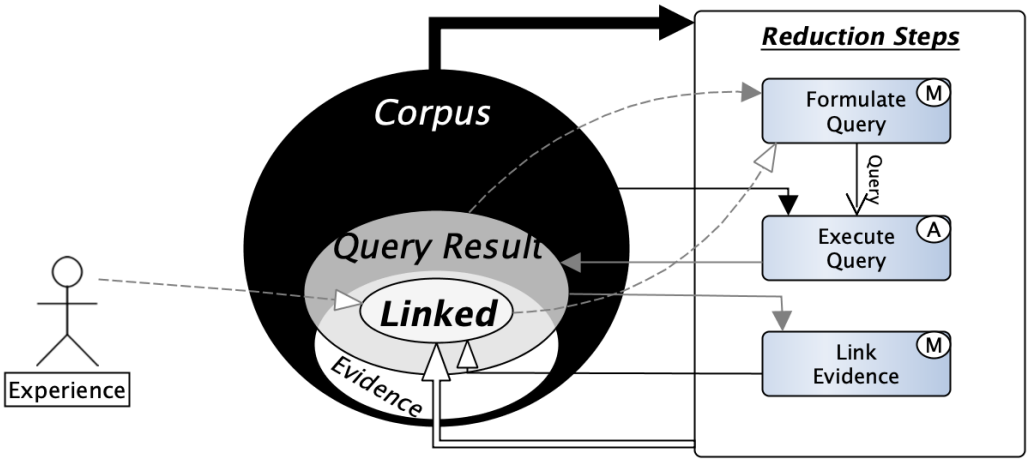

Figure 6.2 – Manually ('M') or automatically ('A') executed steps reduce a corpus to linked evidence. Resources are related to steps by input and output edges, whose color hints at whether it is unknown (black), query-related (gray), or linked (white).

does not represent common usage. It results in an execution of the methodology where only very few existing textual explanation and code example can be linked.

The methodology by its use of queries and tracking those as much as the *linked evidence* helps in raising the confidence in a technology model. We assume that systematically interconnecting many resources is more robust and may help with deriving a technology model more quickly, when compared to an unstructured or biased approach. Bias may be introduced by using a non-representative corpus of resources. Thereby, a construction process is prone to errors, in particular, misconception (See Section 6.4).

The degree of manual effort depends on the given *experience*. If concise textual explanations and idiomatic code examples can be linked and documented 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 can be found, 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 common code examples that are selected from refined query results and then linked. To assure the reproducibility of the construction process, queries are shared as well as the links. In the first case study, 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 methodology in Figure 6.2.

Overall, the methodology can be executed with any set of tools that can be used to systematically perform queries in a text- and code-based corpus. For illustration, we cover different tools to execute queries in literature as well as software projects. To query for code examples, we present two different tools. For the technology model in the first case study, we use the query engine that is developed in [Härtel et al., 2018b]. In the second case study (Section 6.4), we recover code examples by using the GitHub search API. To query for textual explanations, we demonstrate the use of basic tools such as *CTRL+F* in the case studies. In the future, we further investigate on the use of more sophisticated natural language processing tools using natural language pattern [Petrosyan et al., 2015].

6.2.3 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 [Ben-Ari and Yeshno, 2006]. 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 provided 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 [Robillard, 2009].

Teachers (as in teachers wanting to cover EMF in our case) execute the methodology and communicate their findings. This group also includes code reviewers [Bacchelli and Bird, 2013], 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 as well.

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.

6.2.4 Querying for Textual Explanations

A common measure to communicate the trustworthiness of information is to link referential literature as *evidence*. Here, we discuss the use of scientific as well as non-scientific literature. We refer to the latter as developer literature in order to emphasize the main audience.

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 on different technologies, e.g., in [Schauss et al., 2017], we processed referential documentation on diverse metaprogramming technologies. We are also inspired by the search for textual explanations on API types [Petrosyan et al., 2015]. Overall, the goal is to reduce the *corpus* to a small set of concise *textual explanations*, subject to steps as follows:

- *Formulate Query:* A query can either be a search string or an NLP-based algorithm that relates to (parts or synonyms of) the name of an artifact type or relation.
- *Execute Query:* Basic tools for using a search string are *grep* or *CTRL+F*. The reduction done by query execution is facilitated by structural elements in literature. Structural elements can be used as the input. For example, a glossary 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 and may even explicitly name an artifact type or relation of the technology model.

- *Link Evidence:* A linked *textual explanation* potentially encompasses multiple text passages and visual diagrams from which knowledge can be inferred. This complexity should be avoided whenever a more concise *textual explanation* can be linked instead. A concise explanation is always preferable over an ambiguous text.

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. Here, the goal is to reduce a corpus of scientific publications, for example, in Google Scholar, to papers that explain an artifact type or relation. For scientific literature, the reduction steps can be compared to those of a systematic literature study [Kitchenham et al., 2009]. Thus:

- *Formulate Query:* A search string is formulated that relates to (parts or synonyms of) the name of an artifact type or relation. Furthermore, multiple artifact types and relations can also be covered with the same search string.
- *Execute Query:* The search string is sent to a query engine. In Google Scholar, for example, query results are already ranked according to relevance: '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]
- *Link Evidence:* A reasonable subset of papers can be taken from the top results, for example, from Google Scholar. This set of query results can be steadily increased by considering additional results or by adapting the query. When only text passages in the papers are shared, reduction steps are performed in analogy to *developer literature*. Linking one precise textual explanation in 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 [Ed-Douibi et al., 2016]. Papers that explain adapted technology usage are not useful for teaching common technology usage. Thus, exclusion criteria may be identified in this manner.

6.2.5 Querying for Code Examples

The exhaustive recovery of code examples is motivated in our previously conducted analysis of technology usage in Chapter 5. Within the context of a systematic reduction, a corpus of software projects is determined. Queries are formulated and executed to reduce the amount of manually inspected code examples. Here, queries are tested for demo projects, which leads to less manual effort for wild projects.

Demo Projects

Developer literature is often accompanied by demo projects. A demo project serves as a sandbox for testing queries, because it has a simple structure and logic. Hence, the goal is to correctly reduce the corpus of selected demo projects to all code examples for the technology model. This effort is feasible, because it facilitates the reduction of larger corpora, i.e., large wild projects, to code examples. We suggest these steps:

[3] `https://scholar.google.com/intl/en/scholar/about.html` — Requested March 31, 2022

- *Formulate Query:* Formulating 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. We emphasize the need to formulate queries that recognize *all* code examples within selected demo projects, because these queries can be more reliable when applied to wild projects, which potentially have a more complex structure and logic. Developing a query that correctly recognizes all code examples requires systematic debugging and constant inspection of query results.
- *Execute Query:* Feasible tools to execute queries may range from handcrafted miners using *grep* to rule-based reasoners such as QegaL.
- *Link Evidence:* All code examples in demo projects are linked. If the queries need to be adapted for code examples in wild projects, the links to code examples in demo projects are used for regression testing. This way, we make sure that the adapted queries do not introduce new mistakes.

Wild Projects

Different kinds of repositories exist on GitHub ranging from projects developed by students to repositories maintained by professional developers. GitHub can be queried by using its API to identify a set of promising repositories, where the technology is used [Härtel et al., 2018b, Kolovos et al., 2015]. 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 used for demo projects can be reused. While the queries have been tested against sandbox examples, complex wild projects may introduce peculiarities that lead to large numbers of mistaken code examples. In this case, potential false positive and false negative code examples need to be sampled. Then, patterns need to be determined within the sample to systematically improve the queries. Such debugging is illustrated in Section 6.3. When the queries are refined, the code examples from demo projects as well as the samples from wild projects are used for regression tests. This way, we make sure that queries are improved and not changed for the worse.
- *Execute Query:* To execute a query, the same tools can be used as for 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.

6.3 Reproducible Construction of a Model of EMF Code Generation

Here, we construct a technology model on code generation that includes basic artifact types and relations to get the reader acquainted with the execution of the methodology in detail. The results from executing the methodology for the first technology model are representative for a smooth construction process, where every added artifact type and relation is correct and evidence is identified in every type of resource.

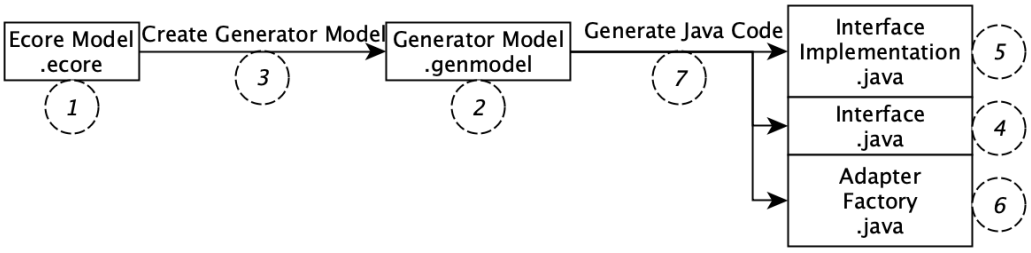

Figure 6.3 – We incrementally construct the technology model from Figure 6.1 in the order that is depicted here (1-7). We present the evidence for it in this Section.

The technology model in Figure 6.3 is inspired by a conceptual model in [Hebig and Giese, 2017]. 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*).

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 dedicated book [Steinberg et al., 2008]. 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 static banner ad on EMF's website[4] that says: 'Buy the book'. Hence, it is recommended by *contributors* of EMF.

Figure 6.4 – 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 Subsection 2.4.4. 'The Generator Model'.

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

[4] `https://www.eclipse.org/modeling/emf/` — Requested March 31, 2022

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

and relation, we identify relevant subsections by running queries against the table of contents. Figure 6.4 presents exemplary titles of (sub-) sections in [Steinberg et al., 2008]. 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'.

An excerpt of a *reduction step protocol* is given in Table 6.1, where each row documents one executed reduction step. In the presented examples, we name the input and output of each executed reduction step to identify the textual explanation for the artifact type *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. This way, the query reduces the book to candidate subsections that are then manually filtered.

The links to textual explanations are accompanied by dedicated rationales in Table 6.2. Here, the evidence is ordered by subjectively estimated importance. Explanations of the artifact type *Ecore model* are introduced in *Subsection 2.3.1*; in the same query results, we also identify *Subsection 12.4.4 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 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 for the fact that a generator model is derived and that the Java code is derived as well. Precise evidence is only 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 configuration.
	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 an 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 the code generation.

Table 6.2 – Types and relations are linked to textual explanations in the book. The quoted text passages and rationales support the links to relevant subsections.

Scientific Literature

We use Google Scholar as the *corpus* that is reduced to textual explanations in scientific literature. Table 6.3 presents the reduction step protocol. Here, we illustrate the transition from a naive query to a refined query. 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 6.3), we only find the paper written by Hebig [Hebig et al., 2012] 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 results and link five results so that artifact types and relations are covered redundantly. The selected papers explain basics of EMF usage and neither present an adaptation of the framework nor explain how to use it together with another software technology.

Table 6.4 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 6.3). 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 link a paragraph or a meaningful figure explained in the paper. The rationales explain why a paragraph or a highlighted figure is linked.

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. [Hebig et al., 2012]	A
3	Link Evidence	*Query results:* See **2**	*Linked*: Hebig et al. [Hebig et al., 2012]	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*: [Kolovos et al., 2010] [Biermann et al., 2010] [Buchmann and Schwägerl, 2015] [Kolovos et al., 2017]	M

Table 6.3 – The reduction step protocol for the scientific literature.

Type/**R**elation	*Links*	*Rationale*
All	[Hebig et al., 2012, Fig. 3]	Uses the same names.
T:Ecore model, T:Generator model, R:Create Generator Model	[Kolovos et al., 2010, Fig. 1]	Text and model refer to: "Ecore metamodel", "GenModel model", "EMF Ecore2GenModel transformation".
T:Ecore model, T:Generator model, R:Create Generator Model	[Biermann et al., 2010]	The introduction explains the dependency of the generator model to Ecore model on the fragment level.
All except -T:Adapter Factory	[Buchmann and Schwägerl, 2015]	'The EMF code generator is always invoked on a so called generator model'
All except -T:Adapter Factory	[Kolovos et al., 2017]	'the GenModel is consumed by a built-in model-to-text transformation'

Table 6.4 – Linked scientific literature that explains the artifact types and relations.

Demo Projects

We adopt the phases suggested by the approach to mining repositories in Section 5.1. Locating and selecting projects as well as detecting instances can be interpreted as reduction steps.

Locating: The EMF book [Steinberg et al., 2008] is accompanied by a set of models that can be downloaded online[5]. When we speak of a demo project, we mean a single EMF project that we built locally based on the set of provided models. The models included in the download are kept in separate folders, where each model and the generated code are described and used in specific chapters of the book. The *Readme* provides the mapping from book sections to the EMF model files.

Selecting: We select the project *PrimerPO* as our *corpus*. It illustrates the basics described up to Chapter 4. If we wanted to focus on advanced usage of EMF, 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, the basic demo project is sufficient.

Detecting: With QegaL, inference of instances of types and relations 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 inferred triple(s) – adding a new triple triggers all rules again. This process stops, when no facts can be added anymore. Table 6.5 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 6.1 + '.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 6.6	M
...	...	...	...	...

Table 6.5 – 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 6.3. Listing 6.1 presents the documented queries that are implemented in QegaL.

[5] `http://www.informit.com/store/emf-eclipse-modeling-framework-9780321331885` — Requested March 31, 2022

```
(?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 6.1 – 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 6.2.

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

Listing 6.2 – QegaL queries to identify code examples for the derivation relation between the Generator model and the Ecore model.

We also use links to code examples that should not result from a query for testing. Such test artifacts are also shared as triples. The principle is exemplified in Listing 6.3. Here, the tests reveal that the '*Package.java' and '*Factory.java' files do not instantiate interfaces, because we interpret the artifact type *Interface* as an interface corresponding to a model class. 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' for creating model instances. More checks on the integrity of query results are explained in the context of querying wild projects.

```
<:/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 6.3 – We test based on false-positive triples.

To identify code examples of artifact types, the type hierarchy of interfaces and interface implementations needs to be resolved. In the QegaL-based implementation, queries identify Java classes and their *extends* and *implements* relationships. To identify code examples of *Interface*, we search for interfaces that (transitively) extend the interface *EObject*. Code examples of *Interface Implementation* are found based on the implementation of already identified examples of *Interface*. At last, code examples of *Adapter Factory* are identified based on the *extends* relation to the interface *AdapterFactory*. Figure 6.5 summarizes the ideas behind the queries to recognize code examples of the relation *Generate Java Code*.

Generator models contain packages and classes in analogy to the generated code. Thus, for every generator package there exists a Java package; for every generator class there exists a Java interface and implementation. A qualified name of the Java package can be derived from the package in the Ecore model. Hence, we first identify the package element in the generator model. This package refers to an Ecore package that delivers the namespace of a package. We built qualified names, such as 'ppo.Item', to identify class decorators in the generator model. For every Java file, we also determine the corresponding qualified names. In

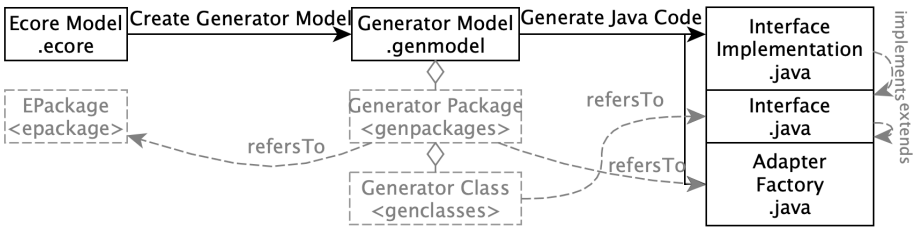

Figure 6.5 – This sketch shows which additional code fragments and references need to be detected to find code examples of the relation *Generate Java Code*.

the end, we match qualified names of generator classes with qualified names of interfaces. At last, interface implementations can then be identified by the implemented interfaces.

Developing a correct query for code examples of the relation *Generate Java Code* is difficult and requires a few iterations. Chapter 4 of the EMF book can be consolidated for the required *experience* to develop the query. We manually persist links to the correct code examples in the demo project first, because we need to assure that all code examples are matched correctly whenever queries are adapted. Table 6.6 presents 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 6.1)
T:GM	`<:/model/PrimerPO.genmodel>`	Generator Model Query (see Listing 6.1)
R:EtoG	(`<:/model/PrimerPO.ecore>`, `<:/model/PrimerPO.genmodel>`)	Foreign Model Query (see Listing 2)
T:Int	`<:/src/ppo/Item.java>,..`	Extends Queries (see Figure 6.5)
T:Impl	`<:/src/ppo/impl/ItemImpl.java>,..`	Implements Queries (see Figure 6.5)
T:AF	`<:/src/ppo/util/PpoAdapterFactory.java>`	Package Reference Query (see Figure 6.5)
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 6.5)

Table 6.6 – 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.

Wild Projects

Based on an investigation in wild repositories, we gain insights into whether the technology model in Figure 6.5 is commonly instantiated in terms of code examples. We again quickly step through the mining process from [Härtel et al., 2018b]. *Locating:* We initiate the reduction by considering all repositories on GitHub. Then, the first query aims to identify repositories that use EMF. *Selecting:* Many repositories exist that use EMF to varying extent. Here, we focus on "vanilla" EMF usage [Härtel et al., 2018b]. Additionally, we exclude repositories 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. *Detecting:* We apply the rules that we have tested for demo projects. *Results:* Table 6.7 summarizes this initial reduction that results in the top ten repositories selected based on their star rating.

ID	*Step*	*In*	*Out*	*Automation*
1	Formulate Query	*Expertise*: What co-technologies distort vanilla usage	*Query*: QegaL query described in [Härtel et al., 2018b] 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 6.8	M
4	Execute Query	*Corpus Resource*: Linked repos from step **3** *Query*: see Table 6.6 for an overview	*Query Results*: see Table 6.8	A
...	...	...	...	...

Table 6.7 – Excerpt of the reduction step protocol for the wild projects. It presents how the set of all repos on GitHub is reduced to vanilla EMF repositories [Härtel et al., 2018b].

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 in a *Generate Java Code* relationship. The query is provided in Listing 6.4. 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 6.8). As a result, the queries that are summarized in Figure 6.5 cannot match any Ecore package. Hence, the relation *GenerateJavaCode* (*R:GtoJ* in Table 6.8) cannot be instantiated (see *R:GtoJ*,

column [3] in Table 6.8). Moreover, we also find repositories, where no model interfaces that extend *EObject* are recognized at all. Still, we focus on idiomatic usage, where a generator model is based on an Ecore model instead of a '.uml' model. Thus, the queries for code examples are not refined.

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

Listing 6.4 – SPARQL queries for sampling suspicious code examples that would lead to more refined queries.

In Table 6.8, 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.

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 6.1)
T:GM	1	2	1	5	2	2	2	1	1	2	Generator Model Query (see Listing 6.1)
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 6.5)
T:Impl	82	2	2	15				46	6	4	Implements Queries (see Figure 6.5)
T:AF	3	1	1	4				1	1	2	Package Reference Query (see Figure 6.5)
R:GtoJ	167	5		38				93	13	4	Reference Query (see Figure 6.5)

Table 6.8 – 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.

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*, where the role of the table of content is emphasized for a simple reduction; in demo projects, code examples are detected manually to then develop queries in a test-driven manner; in wild projects, the queries are executed to find more complex code examples and check their commonness. All textual explanations and code examples can be viewed online.[6]

[6] `https://github.com/softlang/megaemf/tree/master/models/CodeGeneration` — Requested March 31, 2022

6.4 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 6.6 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. According to the resulting megamodel in Figure 6.6, a model class, which is at the level of metamodels, inherits from a class that is part of the metametamodel instead of instantiating it. Scientific literature clarifies the correct alignment of EMF terminology to the common model layers (model, metamodel, and metametamodel) [Bézivin et al., 2005a, Bézivin et al., 2005b, Gascueña et al., 2012]. While executing the methodology, we ignore our experience.

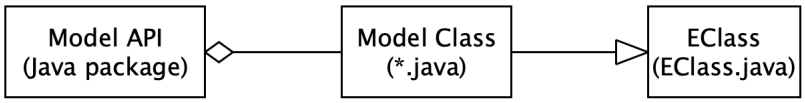

Figure 6.6 – This technology model summarizes a potential misconception on EMF.

Developer Literature

We again consider the EMF book [Steinberg et al., 2008]. *Formulate Query:* We specifically focus on the subtype relation with the queries "extends EClass", "subclass 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 developer literature. Either, the modeled technology usage is too advanced to be covered in the book. Or, it is just not intended by the *contributors*.

Scientific Literature

We could simply refer to [Bézivin et al., 2005a] immediately to reveal the misconception, but we choose to progress by ignoring our *experience*. 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* [Saile, 2010]. In analogy, only one result is returned for "subclass of EClass", a PhD thesis [Rivera, 2010] formalizes laws between instances of *EClass*. For the third query "subtype of EClass", four results are returned. *Link Evidence:* From the last query we explicitly pick papers on EMF profiles [Langer et al., 2011, Langer et al., 2012]. They describe how EMF is adapted in order to introduce *stereotypes* as a concept at the metametamodel layer.

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.

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[7], where an interface called *Node* extends *EClass*. In this repository, not every interface extends EClass. A subtype of 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 [Langer et al., 2011, Langer et al., 2012]. The repository is not linked in the papers, but there exists a repository with more code examples on EMF profiles for the same user.[8]

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, but we also refer to papers that explain the correct alignment of EMF terminology with the different model layers. Based on the lack of code examples in wild projects, we conclude that the included subtype relation does not represent common usage.

[7] `github.com/cmodw/emf-profiles_testpetrinet` — Requested March 31, 2022
[8] `github.com/cmodw` — Requested March 31, 2022

6.5 Threats to Validity

We executed the methodology described in two case studies. The first case study in Section 6.3 exhaustively describes the execution and debugging steps. It also reflects on refining queries and test strategies. The second case study in Section 6.4 describes how to reveal misconception with arguably less effort. Thus, we essentially answered the research question of Section 6.2 by the provision of a methodology for reproducible construction of a technology model that is interconnected with existing textual explanations and code examples and performing two case studies. 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 6.3. 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 existing 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 paper.) It is harder to find existing textual explanations and code examples on more complex aspects. 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.

6.6 Conclusion

In Section 6.2, 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* in 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. Importantly, the methodology features a reduction procedure that is aimed at minimizing the manual effort for identifying and then linking textual explanations and code examples. It complements existing forms of patterns that assist in understanding complex software [Pescador et al., 2015] as well as analysis of API usage [Ratiu et al., 2008, Roover et al., 2013, Robillard et al., 2013].

Chapter 7

Related Work

Overall, we contribute methodologies and case studies to assist in understanding software languages and software technologies. We have focused so far on discovering and verifying knowledge in terms of textual explanations in literature and Wikipedia as well as code examples in software projects.

In this chapter, we give an overview of related research domains. For each domain, we discuss the link to research contributed by this thesis. In Section 7.1, we cover different types of platforms for maintaining and constructing knowledge resources. In Section 7.2, we cover existing knowledge graphs on a broad range of domains that may be reused when constructing domain-specific knowledge resources. Therefore, i.) we cover other kinds of constructed knowledge resources aside from megamodels and knowledge graphs; ii.) we discuss related approaches to mining facts on software languages and software technologies, possibly based on other original resources such as StackOverflow.

Chapter contribution: This chapter summarizes and extends the related work sections in published contributions [Heinz et al., 2017, Heinz et al., 2019, Härtel et al., 2018b, Heinz et al., 2020]. By splitting background and related work, we draw a line between what served clearly as a foundation and what is similar but not adopted directly. Several directions are picked up again in Section 8.2 on future work. Hence, this presentation of related work can also be seen as a preparation for the future work discussion.

7.1 Knowledge Platforms

In this section, we explain two approaches that implement features to make knowledge resources accessible through the web.

Knowledge portals are web applications that expose knowledge buried in an original web resource. In principle, this is exemplified by the idea of a computer language classification portal [Akinin et al., 2012]. Knowledge portals follow several principles that are described in [Staab and Maedche, 2001, Hogan et al., 2020]. A knowledge portal specializes in a specific domain of interest and provides a graphical user interface for easily accessing domain-specific knowledge. The knowledge portal can be easily extended by users and is adaptable to the original web resource. The use of information retrieval techniques or machine learning can help discover relevant information pieces in a raw source. Additionally, a knowledge portal implements features to make the relevant information pieces more accessible. Schema-related features help with instantiating an ontology and validating the knowledge graph against it. Furthermore, an ontology facilitates the integration of features that allow a systematic exploration of the knowledge graph, such as queries on original as well as inferred knowledge. Contributors are provided with editing features to refine (e.g., by deducing and inducing more facts) and maintain an underlying knowledge graph.

Semantic Wikis are focused on advanced mechanisms to maintain facts in a Wiki in a more systematic manner [Buffa et al., 2008]. Instead of mainly textual contributions, structured data is provided. A page in a semantic wiki revolves around a main concept similar to a page on Wikipedia, but it focuses on a single entity. Metadata, such as properties of the covered concept, is encoded in RDF(-like) languages, which enables the use of structured queries and reasoning. Such mechanisms have been elaborated in the *101wiki* to document general concepts exemplified in contributions to *101companies* [Favre et al., 2012b], *101haskell* [Lämmel et al., 2013c], or *metalib* [Schauss et al., 2017], where pages are typed by a namespace, such as *Language*, e.g., for Java[1], or *Technology*, e.g., for EMF[2]. While the benefits of a semantic wiki in the context of teaching are theoretically discussed in [Schaffert et al., 2006], its perceived usefulness is evaluated by students in the context of functional programming lectures in [Lämmel et al., 2013c]. Software languages and software technologies are included as pages in the wikis contributed by [Favre et al., 2012b, Schauss et al., 2017].

7.2 Knowledge Graphs

Several knowledge graphs exist that cover information from a broad range of domains. Here, we summarize their properties. While we have considered them in our research, none of them covered, for example, software languages to a promising degree. They remain potential candidates for knowledge extraction.

WordNet is a graph that consists of words from the English vocabulary and relations of words [Miller and Fellbaum, 2007]. The usefulness of WordNet varies depending on the coverage of terms in a domain. While WordNet is a known assistant in more common knowledge domains, low coverage is reported for more narrow domains in [Wang et al., 2010, Do and Roth, 2012].

YAGO and *Wikidata* are two further examples for knowledge graphs which are more loosely derived from Wikipedia. YAGO [Rebele et al., 2016] is a knowledge graph derived from Wikipedia, WordNet and Geonames that explicitly focuses only on structured knowledge with an underlying ontology, where entities are then again linked by DBpedia as well. Wikidata [Vrandecic, 2012] is largely crafted manually, where the goal is to create a clean structured knowledge graph from scratch.

[1] `https://101wiki.softlang.org/Language:Java` — Requested March 31, 2022
[2] `https://101wiki.softlang.org/Technology:EMF` — Requested March 31, 2022

The Microsoft Academic Knowledge Graph [Färber, 2019] provides entity embeddings for 210 million represented publications. The goal is to assist with entity-centric exploration, data integration and data analysis which includes knowledge discovery. A criteria-based evaluation is used to assess prominent knowledge graphs including DBpedia, Freebase, OpenCyc, Wikidata and YAGO in [Färber et al., 2018]. A type encompassing the term software language (except for computer languages in Wikidata) is not maintained by any of the mentioned resources.

7.3 Knowledge Artifacts on Software Engineering

In the following, we describe several exemplary knowledge resources that cover concepts from the broad domain of software engineering. The scope of covered concepts differs from the scope of our research, but similar methods are used for construction and the purpose of assisting at understanding a broad software engineering concept in a more systematic manner is typically stated as a goal.

7.3.1 Software Engineering Taxonomies

Taxonomies represent a hierarchy of concepts for a target domain [Lämmel et al., 2013a]. We discuss four examples for software engineering taxonomies that were created based on varying methodologies.

In [Forward and Lethbridge, 2008], a taxonomy for the classification of software is proposed. The taxonomy is seeded based on existing taxonomies and then challenged and revised based on a survey. The resulting taxonomy contains categories such as 'Data-dominant software' with 'Personal management' as an exemplary subcategory. It is evaluated by comparing it to the ACM classification system. The '2012 ACM Computing Classification System'[3] is used to provide the hierarchy of research domains in computer science. This way, it facilitates searching for articles that cover similar problems. In [Novak et al., 2010], a taxonomy of concepts for technologies that facilitate static code analysis is presented. The taxonomy is developed by systematically enumerating features that are supported by static code analysis tools. It is evaluated by classifying and comparing four open source tools for static code analysis. In [Wang et al., 2012] a taxonomy of software engineering terms is built based on projects hosted on Freecode, which is a project hosting site that allows tagging software projects. The terms are clustered based on their co-occurence as tags for projects. The taxonomy is evaluated through a user study.

7.3.2 Software Engineering Ontologies

Various ontologies already exist in the field of software engineering. The discussed ontologies are typically evaluated by using them as part of a software system, by exemplary instances, or informal feedback systematically collected from stakeholders.

SWEBOK is a larger effort to collect and structure concepts in software engineering. Ontologies targeting the domain of software engineering and software technology are discussed in [Calero et al., 2006]. In the third chapter [Abran et al., 2006], central principles behind the developments of the ontology for the *software engineering body of knowledge (SWEBOK)* are explained. The SWEBOK ontology has five different objectives: a consistent view of software engineering, a definition of the boundaries of the term software engineering, a characterization of the contents in the software engineering domain, accessibility of SWEBOK's content, and a foundation for a software engineering curriculum that aside from study paths takes certification and licensing into account. Three phases are mentioned towards a validated ontology: '(1) Proto-ontology construction. (2) Internal validations cycle. (3) External validation (and

[3] `https://www.acm.org/publications/class-2012` — Requested March 31, 2022

possibly extension) cycle.' In the current state, all of our contributions have passed the first and second phase. Hence, the third phase on external validation remains as future work.

There exist several ontology engineering applications that share our goal to assist with understanding software engineering concepts. A software component ontology is developed in [Oberle et al., 2004] with the goal of assisting at understanding software components. In essence, the documentation of a software component is enriched through semantic metadata with varying concerns: dependencies to libraries and their versions that allow the detection of inconsistencies; software licenses in use; component capabilities in terms of supported general operations and process such as transactions, classification by a common service taxonomy to group software components based on offered services. The ontology is evaluated by instantiating it using various examples from the J2EE Pet Store Demo. The work is continued as follows. A (core) ontology of software components is developed in [Oberle et al., 2006, Oberle et al., 2009] together with specific ontologies for web services and software services[4]. There, several upper ontologies are systematically reused, namely, *DOLCE* [Gangemi et al., 2002], *Descriptions & Situations (DnS)* [Gangemi and Mika, 2003, Gangemi et al., 2004], *Ontology of Plans (OoP)* [Gangemi et al., 2004], *Ontology of Information Objects (OIO)* [Gangemi et al., 2004]. Furthermore, a core ontology on software (*CSO*) is developed to gather fundamental concepts. In [Kitchenham et al., 1999], an ontology to assist in understanding software maintenance as a task is manually constructed. The result is a framework that helps with categorizing empirical studies by the several influence factors. This way, the ontology helps with reporting, understanding, comparing and reusing empirical results. The ontology is evaluated based on validation by examples, where two different maintenance scenarios are discussed along the ontology. A large set of ontologies is used to structure software maintenance concepts in [Ruiz et al., 2004], where the ontologies are based on experience and collaboration with industrial partners. The set of ontologies is used in a knowledge management system and hence evaluated in terms of Action-Research. Action research is a two stage research validation method. In the first stage, a situation is collaboratively analyzed with the stakeholder. It is understood and reported in terms of a formal model. In the second stage, the situation is changed. This way, research is supposed to transition to practical problems [Avison et al., 2017]. In [de Almeida Falbo et al., 2005] an ontology is developed to assist with knowledge integration in software processes. The authors propose three interlinked ontologies on activities, procedures, and resources in order to facilitate the communication between developers in the setting of a meta-environment that is tailored towards sharing and reusing knowledge artifacts. The ontology is implicitly evaluated, because it is integrated into ODE [de Almeida Falbo et al., 2005] and serves as an execution artifact.

Other ontology engineering approaches share methodological concerns such as the corpora that they aim to reuse or even contribute to. *ALIGNED* is introduced as a suite of ontologies in [Solanki et al., 2016]. It puts emphasis on the connection between the fields of software and data engineering. The approach largely reuses and contributes to *DBpedia*. This suite contains software engineering-specific knowledge, e.g., on the software lifecycle[5] as well as on design intent, data engineering, unified quality reports, domain data models, enterprise information processing, E-research in the Social Sciences and Humanities, Crowd-sourced public datasets, and enterprise software development. *ALIGNED* is used to improve the legal information platform JURION with the goal to structure and facilitate the collaboration between data engineering and software engineering. Hence, its goal is to help data engineers and software engineers to better understand each other. The Software Ontology *SWO* formalizes concepts of software in the domain of biomedical data analysis [Malone et al., 2014]. It, for examples, enables capturing the version and hardware, on which the software was run, in order to assure reproducibility. The ontology's competency is evaluated by informal feedback from

[4] `http://km.aifb.kit.edu/sites/cos/` — Requested March 31, 2022
[5] `http://aligned.cs.ox.ac.uk/ont/slo.html` — Requested March 31, 2022

users that browsed the ontology. Its coverage is evaluated against a list of software extracted from literature. The upper ontology of software models *Unified Software Modeling Ontology (USMO)* [Bräuer and Lochmann, 2008] enables the integration of different DSLs based on their logical foundation. Several implications from open-world assumptions in OWL are addressed. The contributed ontology is manually constructed based on a careful analysis of existing ontologies in the domain of conceptual modeling. It is used to validate the integrity and consistency of its instances based on a reasoner.

7.4 Software Artifacts

In the following subsections, we identify three research directions that focus on software artifacts. They are rather specific to software engineering but may still be considered for ontology engineering with respect to software languages and technologies.

7.4.1 Software Engineering Tutorials

A software engineer tends to use a tutorial to understand how to use a technology. Conceptual knowledge on how the different types of artifact are related is often buried in the textual explanations. As mentioned in Chapter 6, tutorials can vary in their quality, which is an aspect that has been covered in [Lount and Bunt, 2014]. In [Arya et al., 2020], sentences from tutorials are matched with reference developer literature. Seven different degrees of matching are described. For a sentence in a tutorial, a matched sentence in developer literature may be identical, nearly identical, equivalent in terms of their meaning, of one of three different kinds of similar, or unmatched.

At best, actions described in a tutorial can be manually (or automatically) reproduced without issues. An approach and the problems occurring for translating a tutorial to an executable script have been discussed in [Heinz et al., 2016], in particular, for deployment technology. A tool for replaying a tutorial with regard to IDE interactions such as clicking in the context of Eclipse usage is presented in [Zhang et al., 2010]. Recently, the topic of analyzing tutorial videos has received more attention [Ponzanelli et al., 2016, Parra et al., 2018], where the actions are reproduced as results of an action extraction.

7.4.2 Software Artifact Theory

We identify papers contributing theoretical discussions on software artifacts. While a type system for artifacts is defined in [Vignaga et al., 2013] that is tailored towards classifying artifacts common in model-driven engineering, we aim at typing any software artifact, e.g., by its language [Favre et al., 2012c], or its role in technology usage [Heinz et al., 2020].

In [Fernández et al., 2019], the classification of software artifacts is discussed in more general terms. Here, three different perspectives on artifacts are discussed. First, any artifact has a *physical representation* when it is deployed, e.g., a PDF file. Second, any software artifact has a *syntactic structure*. In the case of programs, the structure is defined by a grammar. Hence, programs are typically structured in terms of a tree. For what is commonly referred to as a document, the structure is given through means like a table of content or paragraphs. Third, an artifact has a semantic meaning. It is meant to exist in a context, in which it can be interpreted correctly.

7.4.3 Multilevel Models

In a series of works, Thomas Kühne and Colin Atkinson deal with the problem of multiple levels by categorizing models in the technological space of model-driven engineering [Atkinson and Kühne, 2007]. Here, conformance, instantiation and abstraction are the most essential

relations inspected for models. Instead of only differentiating between two levels, i.e., instance and schema, multiple levels of instantiation are proposed in terms of multilevel models. Principles for maintaining multiple levels of instantiation are discussed in [Atkinson et al., 2014, Atkinson and Kühne, 2015, Kühne, 2018]. Tools to construct and maintain multilevel models are introduced in [Gerbig et al., 2016, Atkinson and Gerbig, 2016, Lange and Atkinson, 2018]. In this context, the authors also discuss the relation of models and ontologies [Atkinson and Kühne, 2016, Kühne, 2016]. We can relate to the idea of multiple levels in the context of models of technology usage. A technology-specific subset of artifacts, e.g., Java code generated by EMF, may be represented by a type on the middle level of abstraction, e.g., *Ecore Model*. At the lowest level, there is the actual code example. The code example can be linked by a reference. It inherits properties, such as the language asserted already for the middle type, and still instantiates the type *Artifact*. We assume that more analogies also with respect to the terms *potency* and *order* can be uncovered when inspecting multiple levels of *subsetOf* relationships, but we leave any deeper discussion and findings to more dedicated future work.

7.5 Empirical Usage Investigations

In this thesis, we have described approaches for large scale empirical analysis on software languages and technologies on Wikipedia and Github Repositories. In the following, we mention related work that uses similar methodologies but aims at different goals for better understanding software languages and technologies in the domains of API Usage, Language Usage, and Developer Profiling.

7.5.1 API Usage Analysis

The popularity and growth rate of testing-related software technology on GitHub is analyzed in [Zerouali and Mens, 2017]. Here, the focus is on frequent co-existence as well as replacement of one library by another. Issues with API usage in mobile applications are discussed in [Lyu et al., 2017]. The authors empirically analyze one thousand applications from Google App Store. They inspect the usage of local databases, which comes with many pitfalls that can lead to high energy consumption and performance problems. Findings show numerous instances of uncommon usage of an API. API caveats that deal with how not to use an API are also discussed in [Li et al., 2018]. There, an approach to recover caveats from unstructured Q&A discussions on Stackoverflow is proposed. The authors mine natural language pattern, as in 'Don't use a HashMap if you are going to have multiple threads'. API elements are recognized based on named entity recognition. More specifically, an improved tokenizer for software-specific vocabulary is used.

Pattern of API usage are recovered in [Saied et al., 2015] by detecting frequent co-usage relationships between API methods. A hierarchical clustering approach is used to distill the core of a pattern. Some API methods may be optional in specific scenarios. Thus, isolating groups of methods provides deeper insights into API usage. In [Saied et al., 2018], the approach is extended to identifying groups of API methods over multiple libraries.

In [Härtel et al., 2018a], APIs are classified by their programming domain (such as GUI programming, database programming, etc.). The authors evaluate different hierarchical clustering parameters, for example, the analytical approach to topic clustering that is based on Inverted Document Frequency (IDF), Latent Semantic Indexing (LSI), or Latent Dirichlet Allocation (LDA). Here, the names of methods in an API constitute a document. Different parameter configurations are explored and compared against a base line from previous work as well as the category system on *Maven Central*.

In [Eilertsen and Bagge, 2018], the co-evolution of APIs and client code is discussed. When an API changes, client code needs to be adapted as soon as the dependency is updated.

Additionally, frequent patterns in client code can trigger a change in APIs. Hence, the authors state that better understanding API usage in terms of capturing the relation between API and client code is crucial and needs to be addressed in more explicit ways.

7.5.2 Language Usage

In literature, we find many empirical studies focusing on Java repositories. The usage of *GoF* design patterns in repositories on Source Forge is analyzed in [Hahsler, 2003]. The authors preselect 1319 projects using Java, where only 128 projects use design patterns at all. The results show that the Command pattern is the most popular pattern. In [Baxter et al., 2006], the structure of Java software in the wild is analyzed. The authors manually collect a corpus consisting of 56 applications written in Java. Metrics are computed, such as the number of interfaces implemented in the application. For a subset of the metrics, evidence is found that they obey a power law. Pattern of usage for the Java streaming API, such as the use of parallel streams, are analyzed in 34 Java projects in [Khatchadourian et al., 2020]. An investigation of common mistakes that lead to bugs motivate discussion on best practices and anti-pattern.

A survey on empirical studies on language usage in [Favre et al., 2011] lists related work according to the analyzed language, where not only repositories using Java are analyzed. An empirical investigation on the usage of graph query languages on GitHub is conducted in [Seifer et al., 2019]. The investigation inspects GitHub projects using SPARQL, Cypher, Gremlin or GraphQL. For SPARQL and Cypher, evolutionary aspects of the projects are inspected. The investigations show that more SPARQL artifacts are edited but Cypher is gaining attention. More empirical research on GitHub can be found, for example, on COBOL [Chevance and Heidet, 1978], XSD [Ralf Lämmel and Stan Kitsis and Dave Remy, 2005], UML [Robles et al., 2017] Python [Malloy and Power, 2019], Haskell [Lima et al., 2016], multiple programming languages in a repository [Mayer and Bauer, 2015], or general language popularity [Meyerovich and Rabkin, 2013].

Corpora for empirical analysis are also based on sources other than project hosting sites. In [Lämmel and Pek, 2013], empirical analysis is tailored towards better understanding usage of the policy language P3P[6] in detail, where an extracted corpus is based on mining the *Open Directory Project (ODP)* as the source for finding websites with policies. 6,182 policy files are mined. The authors present findings about vocabulary, identified validation processes, clones, and a range of metrics. Questions on StackOverflow related to the programming language Swift are mined in [Rebouças et al., 2016]. Furthermore, the authors conducted interviews with Swift developers to find out whether StackOverflow questions reflect the most essential problems. Their findings show that the questions cover understanding the syntax of the language and also relate to issues with central software technology such as Xcode, which is the IDE for developing Swift applications.

7.5.3 Developer Profiling

Aside from empirical investigations on software projects, the problem identifying experts for a software language or software technology is related. Problems range from identifying the expert for fixing a bug [Anvik et al., 2006], matching skills of developers on GitHub to job advertisements [Hauff and Gousios, 2015] with a focus on software language used and library dependencies in software projects, identifying experts on StackOverflow [Huang et al., 2020], and determining technical as well as application domain experts from source code [Sindhgatta, 2008] by processing linguistic information similar to [Ratiu and Deissenboeck, 2006] but identifying discriminating terms without matching them to WordNet.

[6] `https://en.wikipedia.org/wiki/P3P` — Requested March 31, 2022

7.5.4 Recommender Systems

A recommender system can leverage available data and factor crowd contributions into a recommendation of a knowledge resource as exemplified by the following referenced work. To facilitate the search in Stack Overflow, tags are used for categorization and to facilitate the discovery of knowledge on software technology usage. A recommendation system can be used to propose tags for an article [Wang et al., 2018]. Furthermore, Stack Overflow posts can also be recommended based on the similarity of task descriptions as available in user stories, for example, in Jira [dos Santos et al., 2019]. Recommender systems can also be used to improve the learning experience in Wikipedia by recommending related articles [Schwarzer et al., 2016], and even improve structured information in infoboxes [Bostandjiev et al., 2011]. With regards to the usage of APIs, reference literature can be recommended based on word patterns [Robillard and Chhetri, 2015]. Additionally, pattern of API usage can be recommended based on the co-existence relation between method calls of an API class [Niu et al., 2017], which reflects common usage of an API.

7.6 Natural Language Processing

Facts can be recovered from linguistic information available in knowledge resources like Wikipedia [Flati et al., 2016] as well as programs [Ratiu and Deissenboeck, 2006, Sindhgatta, 2008]. In this section, we cover techniques used throughout scientific literature for recovering knowledge from text. This overview is loosely inspired by the overview that is given in [Hogan et al., 2020], but it does not aspire to cover the full breadth. We aim to cover techniques in already referenced literature and lead towards *distant supervision* as a promising technique for a continuation of conducted research in [Heinz et al., 2019] as well as [Heinz et al., 2020]. The techniques are ordered by their complexity in the sense that a more complex technique may require the use of preceding techniques.

7.6.1 Term Extraction

The extraction of concepts or identification of topics often requires recovering all words as a set for each document. Not every word is feasible as a concept. Hence, stop-word lists can be used to exclude common English. Stop-word lists may either be provided by software technology such as *CoreNLP* and *NLTK* or gathered based on the analysis of term frequency to exclude common English vocabulary. Different kinds of word normalization are used in literature, namely, *stemming*, for example, based on the Porter Stemmer [Härtel et al., 2018a], and lemmatization [Flati et al., 2014, Flati et al., 2016]. While the Porter Stemmer transforms the word ‘languages’ to ‘languag’ which is the word stem, lemmatization transforms it to ‘language’ which is its canonical form.[7]

7.6.2 Named Entity Recognition

Some terms in a text may not be part of common English vocabulary. They represent a central known concept with a specific meaning. Terms can be matched with entries from lexicons of senses, e.g., WordNet [Miller and Fellbaum, 2007] or BabelNet [Navigli and Ponzetto, 2012]. Other approaches even match with entries from knowledge graphs. For example, DBpedia Spotlight [Mendes et al., 2011] is used for recovering software engineering concepts from job advertisements in [Hauff and Gousios, 2015]. This way, concepts are recognized in text and linked to a DBpedia page and thus to a Wikipedia article. This way, knowledge beyond what is expressed in the text itself is added. Detecting named concepts is referred to as *Named Entity Recognition (NER)*, which is clearly distinguished from word sense disambiguation ,

[7] `https://text-processing.com/demo/stem/` — Requested March 31, 2022

which is also called Named Entity Disambiguation (NED), e.g., in [Hauff and Gousios, 2015]. A dedicated approach to named entity recognition for software engineering vocabulary is developed in [Li et al., 2018].

7.6.3 Part-of-speech tagging

A word can be annotated with a part-of-speech tag, such as noun (NN), verb (VB), or adjective (JJ)[8]. POS-tagging is used in [Falleri et al., 2010] to extract hypernym relations between identifiers in a program. Whole sentences with each word with such a tag can be used to extract semantic relations based on the order of words, i.e., a Hearst pattern [Hearst, 1992]. Manually crafted rules are used due to their more controllable and predictable behavior [Chiticariu et al., 2013]. For hyponym extraction [Hearst, 1992], the type is given and the instances or subtypes are extracted. The contrary is hypernym extraction, where the type is what is identified in an automated manner. An exemplary pattern from [Hearst, 1992] is: *such NP as {NP,}* {(or | and)} NP*. It can be used to recover the instance relationship in '... works by such authors as Herrick, Goldsmith, and Shakespeare.' Another example from [Hearst, 1992] is: *NP {, NP}* , or other NP*. It can be used to recover the subtype relation in 'Bruises, wounds, broken bones or other injuries...'. Such Hearst-pattern can be learned [Roller et al., 2018]. Aside from such Is-a relations, part-of relations can be extracted as well [Arnold and Rahm, 2015].

7.6.4 Dependency trees

Facts such as *is-A* relationships can be extracted based on the grammatical structure of a sentence. Such extraction does not only take the word order and part-of-speech tags into account but grammatical relations between words as well. Such lexico-syntactic relations are extracted based on a model of a specific language that is readily provided by technology such as Stanford CoreNLP [Manning et al., 2014]. Hypernyms are extracted based on the dependency structure, e.g., in [Flati et al., 2014, Flati et al., 2016].

7.6.5 Relation Extraction

Hypernym recovery is a specific form of *relation extraction*. Pattern in natural language statements are identified and used for mining ontological relations. Relation extraction is a field with many approaches that range from manually crafted pattern [Hearst, 1992, Arnold and Rahm, 2015, Flati et al., 2014, Flati et al., 2016, Heinz et al., 2019] to supervised methods that require a lot of labeled data [Wu and Weld, 2008] and unsupervised methods [Roller et al., 2018], also referred to as *Open Information Extraction* [Lu and Du, 2017, Wu and Weld, 2010]. An example for a more elaborate technology used in industry with a dedicated language to specify pattern over linguistic structure is given in [Chiticariu et al., 2018]. An approach to relation extraction based on bootstrapping that is uses *Stanford Core NLP* is presented in [Gupta and Manning, 2014].

For relation extraction, clean natural language text is required. As Wikipedia markup is semi-structured and the markup's grammar is context-sensitive [Zaytsev, 2011], it typically needs to be preprocessed. However, markup can be useful for extracting facts from semi-structured sources, where it is exploitable [Presutti et al., 2014, Lockard et al., 2018, Lu et al., 2013, Martínez-Rodríguez et al., 2020].

[8] `https://corenlp.run/` — Requested March 31, 2022

7.6.6 Distant supervision

Distant supervision aims to solve the problem of a limited amount of training data for relation extraction [Mintz et al., 2009]. It uses seeds extracted, for examples, from a knowledge base such as Wikidata. A seed example consists of two recognizable entities that instantiate the relation that is supposed to be extracted. Any sentence in the corpus that contains the two entities is likely to encode the relation as well. Hence, in the approach, every such sentence is considered as positively labeled training data. Then, negatively labeled training data is gathered by randomly sampling sentences excluding the positively labeled ones. This way, an actual false negative may be included in the training data, but the mere amount of negative examples mitigates any effect.

At last, the training data can be fed to a readily available relation extraction tool, such as Stanford Relation Extractor [Manning et al., 2014]. The simplification resulting from considering all sentences with the two recognized entities as positive sample is further discussed in [Riedel et al., 2010]. A modification is introduced in [Aprosio et al., 2013], where the recovered samples are again labeled as positive or negative by using several heuristics that aim to differentiate between whether they actually express the relation or not.

Chapter 8

Conclusion

Software engineers deal with a myriad of software languages and software technologies. Information on when and how to use which in what scenarios is scattered over many heterogeneous resources. Furthermore, software languages and software technologies are strongly coupled. Software technology usage typically involves a set of heterogeneous languages and vice versa. The usage of a software language or a software technology has implications on the layout and structure of a software system. We assume that this coupling adds to the difficulty of understanding idiomatic usage. In this thesis, we contribute approaches that deal with discovering and structuring knowledge resources while assuring quality, and making interconnected resources of knowledge more accessible. This way, we aim to assist with understanding the tools with which any software system is engineered, i.e., software languages and software technologies.

1. A core ontology of a language-centric perspective on artifacts, i.e., linguistic architecture, is contributed in Chapter 3 as a continuation of research on linguistic studies in software engineering [Favre et al., 2011, Favre et al., 2012c, Lämmel et al., 2014a, Härtel et al., 2017].

2. An approach to discovering common knowledge on software languages on Wikipedia is contributed in Chapter 4, which hence makes use of one of the most promising sources for knowledge extraction of software engineering concepts today [Robillard and Treude, 2020].

3. An approach to empirically analyzing pattern of technology usage is summarized in Chapter 5 with a case study on EMF as the contribution of this thesis.

4. A methodology to the reproducible construction of conceptual overviews of complex technology usage as opposed to API usage [Roover et al., 2013, Robillard and Chhetri, 2015] is presented in Chapter 6.

8.1 Discussion

In this section, we summarize our contributions and answers to each research question. For each contribution, we repeat the research questions, name the applied methodology and summarize major threats to validity again while proposing how the threats can be addressed in the future.

8.1.1 Axioms of Linguistic Architecture

As the first contribution, we provide surveying as well as formal insights on language-centric documentation of technology. We have answered the following research questions.

RQ A1: What types of entities and relations are common in megamodels?

RQ A2: What modeling idioms exist for (language-centric) megamodels that can be formalized as axioms?

Methodology & Results

As the methodology to answer the first question, we have executed a systematic literature study in the megamodeling domain. The results show that several entity types and relations are common in megamodels, e.g., conformance. For the second research question, we present formal axioms and discuss them along examples on an EMF project.

Threats to Validity

There are several threats to validity for our research. First and foremost, the alignment of vocabulary recovered from the set of analyzed papers was conducted manually and only by us. Hence, we cannot guarantee that the alignment is correct and that we did not miss any hidden semantics. The correct alignment could only be achieved by a dedicated collaboration with the respective authors in the domain. Second, our alignment is inspired by the vocabulary that was introduced throughout our research and is hence biased by our modeling experience. Third, the axiomatization is subjective as its origin is expertise from our research group and research fellows. It requires further examples from a broad range of programming domains constructed by other software engineers in order to be validated more thoroughly. Nevertheless, as mentioned in Section 7.3, this challenge is to be expected according to the development stages described for *SWEBOK* in [Abran et al., 2006], where the prototypical construction and internal validation cycles are succeeded by an external validation.

8.1.2 Software Languages at Wikipedia

As the second contribution, we provide insights on how to discover common knowledge on software languages at Wikipedia. We have answered the research questions below.

RQ W0: What types of content in a single Wikipedia article can be used for extracting knowledge and what challenges need to be addressed?

RQ W1: How can we classify Wikipedia articles by their relevance to a given domain when relevant articles are rare and multiple main topics are covered by articles?

Methodology & Results

We have pointed out how Wikipedia's author guidelines challenge knowledge discovery. Nevertheless, we develop a methodology, which is evaluated in a case study on discovering software language articles. We propose how to deal with a domain in which relevant articles are rare events. The evaluation data is recovered from a survey, in which 31 fellow researchers participated. Based on the learned decision tree, we report what content features indicate a software language article, which may also be seen as a hint towards what types of structured content should be in any representative software language article.

Threats to Validity

The methodology itself has only been applied to the domain of software languages. More case studies are required to show and explain when it is useful. As a methodological concern,

we also have not considered an analysis of whether any categorical features correlate. A correlation can have an influence on the learned decision tree.

The results of the case study may be subjective, even though we use labels for each article by two different researchers. The study in [Combemale et al., 2017] discloses differences in opinions on what a software language is. Unclear corner cases are also discussed in the context of our case study in Section 4.3.2. To resolve this issue, collaborative efforts are required to investigate especially on corner cases without consensus, possibly within the context of SLEBOK [Combemale et al., 2017].

Therefore, we may have trained a classifier based on subjective opinions on what a software language is. Even though, we offer a definition for the class, we cannot guarantee that it was followed or understood. Hence, to reach an improved understanding, *competency questions* as already sketched in Chapter 3 require further investigation. We hypothesize that in different contexts, different *competency questions* can be stated for the ontological class *software language*. Hence, a feasible effort may be to ask experts what competency questions they would state. This way, collaborative construction of an ontology would be simulated for a single class [Sure et al., 2002].

8.1.3 Technology Usage Mining

In Chapter 5, we discuss a large-scale empirical study on patterns of technology usage by mining repositories on GitHub [Härtel et al., 2018b]. Thereby, we answer the following research question.

> *RQ T1: How can we locate traces of technology usage on GitHub?*

Methodology & Results

We develop and use incremental fact extraction from the GitHub repositories that are identified as promising resources for analyzing technology usage. For a chosen technology, such as EMF in the case study, the steps involve i.) developing pattern of technology usage, ii.) locating candidate repositories, iii.) selecting repositories according to criteria, iv.) detecting instances of developed pattern, and v.) reporting results. As a result, we have developed an emerging catalog of EMF usage pattern and analyzed their usage in 1438 selected repositories from initially located 5759 EMF repositories.

Threats to Validity

As a threat to external validity, we have only analyzed 1438 "Vanilla EMF" repositories. We might miss repositories, which leads to results in terms of the frequency of detected pattern that are not representative for the overall set of EMF projects on GitHub. This can be addressed directly by exploring different configurations for repository selection. Exploring only Maven repositories can lead to different findings. Code may only be generated by a Maven build step and hence not even committed. Therefore, more effort is needed to gain insights independent of build system or even project layout. Although the inference rules for detecting patterns have been tested and systematically debugged, we may also still miss particular instances in unseen repositories.

8.1.4 Technology Models

As the fourth contribution, we discuss reproducibility in the context of constructing technology models as an essential quality factor. We answer the dedicated research questions.

> *RQ T2: How can we construct a technology model in a reproducible manner so that it is interconnected with existing textual explanations and code examples?*

Methodology & Results

As a summary of our experience in modeling technology usage, we propose a query-based methodology for constructing reproducible and interconnected technology models [Heinz et al., 2020]. The interconnection to textual explanations and code examples gives a technology model significant value that is greater than a "meaningless diagram". The first case study puts emphasis on getting the reader acquainted with the methodology by making the execution of the methodology more transparent through exemplary reduction step protocols. Afterwards, the second case study is used to demonstrate how the methodology is robust enough to prevent any kind of misconception at the time of the construction.

Threats to Validity

The most significant threat here is again subjectivity. The authors of [Heinz et al., 2020] are familiar with technology modeling. Hence, there is an experience factor that may influence results of executing the methodology. Furthermore, the authors appear as the experts for EMF and evaluate the quality of linked resources. High quality standards for linked resources can only be gained by efforts of independent experts. Next, the methodology has only been rigorously executed for technology models of EMF code generation. Hence, benefits and feasibility still need to be confirmed for other complex technology usage scenarios. Still, many technology models have been created and published by the research group [Lämmel et al., 2014a, Lämmel and Varanovich, 2014, Favre et al., 2012c, Favre et al., 2012a, Zaytsev, 2012, Schauss et al., 2017], for which the methodology has been executed in simplified ways. Therefore, we have high confidence in the usefulness of the methodology. The benefits of linking technology-specific knowledge to source code have been evaluated in the context of a university course [Lämmel et al., 2013c], but an evaluation of the usefulness of technology models in the context of university or professional class rooms is still abundant.

8.2 Future Work

So far, we have summarized open challenges mainly in the context of threats to validity for each contribution, which can be addressed by further efforts. In this section, we sketch future work directions that are independent of individual contributions and address cross-cutting concerns.

8.2.1 External Validation

All contributions have passed the phases of prototypical construction and internal validation cycles [Abran et al., 2006]. Community efforts are left that help with external validation.

Knowledge Platforms

Community efforts may for example rely on platforms for knowledge sharing in the form of knowledge portals [Staab and Maedche, 2001, Akinin et al., 2012] or semantic wikis [Favre et al., 2012a]. To further validate the published research, especially, the axiomatization of megamodeling vocabulary [Heinz et al., 2017], and technology models of EMF in [Heinz et al., 2020], experts from different research groups need to be involved. This way, the matter of subjectivity on existing models can be resolved as the models may be biased by our expertise and experience. Both contributions contain formal/systematic approaches to mitigate the problem of subjectivity by checking consistency against laws and reference material, but there is no direct way to tell whether other experts would come to the same conclusions. Secondly, the generality and usability still needs to be evaluated by further case studies. Detailed

competency questions that depend on a given context for the term 'software language' help to make an informed decision.

Inter-rater Agreement

A platform that exposes shared knowledge may also expose the degree of agreement on the classification of concepts, which can be expressed in terms of the computed value for *Cohens Kappa* [Landis and Koch, 1977] for two participants, Fleiss Kappa for multiple participants, and *Powers Kappa* [Powers, 2003] that considers known odds. Higher Kappa values mean a high agreement. The usefulness of such values is discussed in [Powers, 2012]. Such measurement is useful when several participants (human or machine raters) answer the same set of classification questions. The measurement allows the identification of problematic classifications that require further investigation and insights. An example, where inter-rater agreement is helpful to identify issues with the notion of a software language is discussed in Chapter 4. While there exist definitions in different resources [Lämmel, 2018, Favre et al., 2009, Kleppe, 2008], there still seems to be no complete consensus, which becomes apparent in surveys, where participants are asked to tell which item is a software language and which is not [Heinz et al., 2019, Combemale et al., 2017] (See Chapter 4). A further investigation here is required that also clarifies the competencies of the ontological class *software language* [Sure et al., 2009]. Competency questions can also be based on investigating into descriptions in different text corpora: What kind of information is given in sentences that include the name of a software language (or subcategories like programming languages)?

EMF Reference Technology Models

This thesis 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. A long 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.

We hypothesize that there exist common pattern of technology usage that are not explicitly documented in referential documentation. They may only become visible based on deduction by combining multiple concepts, such as technology-specific programming techniques and design patterns. A further research effort may focus on revealing such undocumented pattern in a reproducible manner in terms of a technology model that is mainly interconnected with code examples and annotations by multiple independent technical experts. Freely learning pattern without the involvement of domain experts in a fashion that resemble Open information extraction (OpenIE) is an alternative to an expert-based approach [Lu and Du, 2017, Wu and Weld, 2010].

Technology Models across Technology Spaces

Beyond the scope of EMF, we would like to see that the contributions help with broader efforts on developing a comprehensive ontology in software engineering [Cesar Gonzalez-Perez, 2017], and more specifically, the software language engineering body of knowledge (SLEBOK) [Combemale et al., 2017]. To this end, further refinements of the proposed methodology may be needed as well as additional forms of validation. We hypothesize that exploring interconnected technology models with dedicated tool support, for example, similar to [Roover et al., 2013], may help in understanding EMF, and possibly other complex technology, such as *Xtext* or *Apache Spark*. Thereby, we also summarize the remaining research

challenges for the axioms presented in Chapter 3. To properly evaluate the axiomatization in terms of its usefulness and general fit, the static semantics needs to be repeatedly challenged by models on the usage of diverse technologies. On a more foundational level, the axiomatization can also be challenged by analyzing common vocabulary in actual software documentation as found, for example, in README files on GitHub as a follow-up of [Prana et al., 2019].

8.2.2 Relation Extraction from Text

Reading developer literature, i.e., reference literature when available and textual explanations written by the community, is a common task for developers [Robillard, 2009]. Further exploration may uncover how frequently typical documentation idioms from referential documentation are aligned with vocabulary in software ontologies such as SoLaSoTe, i.e., through a corpus-driven evaluation. We hypothesize that more textual explanations can be extracted using known approaches to relation extraction (see Section 7.6). We assume that case studies on extracting textual explanations on technology usage inspired by efforts on API usage [Maalej and Robillard, 2013, Robillard and Chhetri, 2015] may contribute more insights into the nature of technology documentation and helps with identifying relations between heterogeneous types of artifacts in an automated manner.

8.2.3 Improvements on Interconnecting Code Examples

In future work, we would like to address several challenges to interconnecting code examples in megamodels. We envision an analysis on how technology usage changes over time by processing repositories' commit history [Härtel and Lämmel, 2020]. Furthermore, we would like to increase precision with regard to some aspects of correspondence, completeness, and consistency by integrating type resolution with Java classpath recovery. Moreover, we also work on a more profound generalization of referring to and accessing 'arbitrary' code or model elements across technological spaces through uniform resource accessors. Ultimately, we would like to move from small patterns of technology usage to more complex and modular megamodels for technology documentation [Härtel et al., 2017].

8.2.4 Technology Model-based Comparison

Software languages can be used as the central classification dimension for artifacts. Megamodels, e.g., in [Favre et al., 2012c], raise awareness of different kinds of software languages that come with *JAXB* usage (See 2.5 for an example). A subset of Java is identified that is implemented by *JAXB* that relates to all Java code that can be generated by *JAXB*'s compiler. An alternative to using *JAXB* for XML serialization is *Jackson.*[1] While we hypothesize that technology models also help with comparing technologies, there is no supporting case study for this claim. A case study may investigate in which scenarios developers use either *JAXB* or *Jackson*. The Readme for *Jackson* on GitHub already hints at a difference facilitated concepts: 'Further, the goal is to emulate how JAXB data-binding works with "Code-first" approach (that is, no support is added for "Schema-first" approach).' Differences in terms of functionalities may be modeled as feature models for XML processing technologies, which is then instantiated by *JAXB* and *Jackson*. We assume that such a structured comparison may help with deciding for a technology that fits a given scenario.

8.2.5 Megamodeling Transients

In linguistic architecture, transient artifacts are considered as well as artifacts with a persistent manifestation. While transients, such as objects at runtime, have been discussed in theory in

[1] `https://github.com/FasterXML/jackson-dataformat-xml` — Requested March 31, 2022

Chapter 3 and in exemplary models in [Favre et al., 2012c], an investigation is needed that covers the relation between object-data and data encoded in technology-specfic languages, such as the XSD or Java subset implemented by JAXB. Technology models can be constructed in a reproducible manner that involve transients as another type of artifact. Tools different from those used so far are needed for their detection in code through static analysis [Lämmel and Varanovich, 2014, Härtel et al., 2017, Rocco et al., 2017] and debugging approaches. Understanding transients and their implicit dependencies has been identified as a challenge to debugging programs [Parnin and Orso, 2011]. Hence, an investigation into the competency of transient artifact types can be conducted, e.g., by answering the following draft question: Can technology models that contain transient artifact types help with better understanding programs? A study on common problems with (de-) serialization of Java objects can help with answering such research question.

Bibliography

[Abran et al., 2006] Abran, A., Cuadrado, J. J., Barriocanal, E. G., Mendes, O., Alonso, S. S., and Sicilia, M. (2006). Engineering the ontology for the SWEBOK: issues and techniques. In *Ontologies for Software Engineering and Software Technology*. Springer.

[Acosta et al., 2018] Acosta, M., Zaveri, A., Simperl, E., Kontokostas, D., Flöck, F., and Lehmann, J. (2018). Detecting linked data quality issues via crowdsourcing: A dbpedia study. *Semantic Web*, 9(3):303–335.

[Afzal et al., 2018] Afzal, W., Brunelière, H., Ruscio, D. D., Sadovykh, A., Mazzini, S., Cariou, E., Truscan, D., Cabot, J., Gómez, A., Gorroñogoitia, J., Pomante, L., and Smrz, P. (2018). The megam@rt2 ECSEL project: Megamodelling at runtime - scalable model-based framework for continuous development and runtime validation of complex systems. *Microprocessors and Microsystems - Embedded Hardware Design*, 61:86–95.

[Akinin et al., 2012] Akinin, A., Zubkov, A., and Shilov, N. (2012). New Developments of the Computer Language Classification Knowledge Portal. In *Proc. Spring/Summer Young Researchers' Colloquium on Software Engineering*, number 6.

[Anureev et al., 2008] Anureev, I. S., Bodin, E. V., Gorodnyaya, L. V., Marchuk, A. G., Murzin, F. A., and Shilov, N. V. (2008). On the problem of computer language classification. *Joint NCC&IIS Bulletin, Series Computer Science*, 27:1–20.

[Anvik et al., 2006] Anvik, J., Hiew, L., and Murphy, G. C. (2006). Who should fix this bug? In *Proc. ICSE*, pages 361–370. ACM.

[Aprosio et al., 2013] Aprosio, A. P., Giuliano, C., and Lavelli, A. (2013). Extending the coverage of dbpedia properties using distant supervision over wikipedia. In *Proc. ISWC*, volume 1064 of *CEUR Workshop Proceedings*. CEUR-WS.org.

[Arnold and Rahm, 2015] Arnold, P. and Rahm, E. (2015). Automatic extraction of semantic relations from wikipedia. *International Journal on Artificial Intelligence Tools*, 24(2):1540010:1–1540010:36.

[Arya et al., 2020] Arya, D. M., Guo, J. L. C., and Robillard, M. P. (2020). Information correspondence between types of documentation for apis. *Empirical Software Engineering*, 25(5):4069–4096.

[Ashburner et al., 2000] Ashburner, M., Ball, C. A., Blake, J. A., Botstein, D., Butler, H., Cherry, J. M., Davis, A. P., Dolinski, K., Dwight, S. S., Eppig, J. T., et al. (2000). Gene ontology: tool for the unification of biology. *Nature genetics*, 25(1):25–29.

[Aßmann et al., 2006] Aßmann, U., Zschaler, S., and Wagner, G. (2006). Ontologies, metamodels, and the model-driven paradigm. In *Ontologies for Software Engineering and Software Technology*. Springer.

[Atkinson and Gerbig, 2016] Atkinson, C. and Gerbig, R. (2016). Flexible deep modeling with melanee. In *Modellierung Workshopband*, volume P-255 of *LNI*, pages 117–122. GI.

[Atkinson et al., 2014] Atkinson, C., Gerbig, R., and Kühne, T. (2014). Comparing multi-level modeling approaches. In *Proc. MoDELS*, volume 1286 of *CEUR Workshop Proceedings*, pages 53–61. CEUR-WS.org.

[Atkinson and Kühne, 2007] Atkinson, C. and Kühne, T. (2007). A tour of language customization concepts. *Advances in Computers*, 70:105–161.

[Atkinson and Kühne, 2015] Atkinson, C. and Kühne, T. (2015). In defence of deep modelling. *Information & Software Technology*, 64:36–51.

[Atkinson and Kühne, 2016] Atkinson, C. and Kühne, T. (2016). Demystifying ontological classification in language engineering. In *Proc. ECMFA*, volume 9764 of *Lecture Notes in Computer Science*, pages 83–100. Springer.

[Atzeni and Atzori, 2017] Atzeni, M. and Atzori, M. (2017). CodeOntology: RDF-ization of source code. In *Proc. ISWC*, volume 10588 of *Lecture Notes in Computer Science*, pages 20–28. Springer.

[Avison et al., 2017] Avison, D. E., Kock, N., and Malaurent, J. (2017). Special issue: Action research in information systems. *J. of Management Information Systems*, 34(3):630–632.

[Bacchelli and Bird, 2013] Bacchelli, A. and Bird, C. (2013). Expectations, outcomes, and challenges of modern code review. In *Proc. ICSE*, pages 712–721. IEEE Computer Society.

[Bagge and Zaytsev, 2014] Bagge, A. H. and Zaytsev, V. (2014). Languages, models and megamodels. In *Proc. SATToSE*, volume 1354 of *CEUR Workshop Proceedings*, pages 132–143. CEUR-WS.org.

[Barbero et al., 2008] Barbero, M., Jouault, F., and Bézivin, J. (2008). Model Driven Management of Complex Systems: Implementing the Macroscope's Vision. In *Proc. ECBS*, pages 277–286. IEEE.

[Basciani et al., 2014] Basciani, F., Rocco, J. D., Ruscio, D. D., Salle, A. D., Iovino, L., and Pierantonio, A. (2014). Mdeforge: an extensible web-based modeling platform. In *Proc. CloudMDE@MoDELS*, volume 1242 of *CEUR Workshop Proceedings*, pages 66–75. CEUR-WS.org.

[Bassani and Viviani, 2019a] Bassani, E. and Viviani, M. (2019a). Automatically assessing the quality of wikipedia contents. In *Proc. SAC*, pages 804–807. ACM.

[Bassani and Viviani, 2019b] Bassani, E. and Viviani, M. (2019b). Quality of wikipedia articles: Analyzing features and building a ground truth for supervised classification. In *Proc. IC3K*, pages 338–346. ScitePress.

[Bastarrica et al., 2014] Bastarrica, M. C., Simmonds, J., and Silvestre, L. (2014). Using megamodeling to improve industrial adoption of complex MDE solutions. In *Proc. MiSE*, pages 31–36. ACM.

[Baumeister and Seipel, 2006] Baumeister, J. and Seipel, D. (2006). Verification and Refactoring of Ontologies with Rules. In *Proc. EKAW*, volume 4248 of *Lecture Notes in Computer Science*, pages 82–95. Springer.

[Baumeister and Seipel, 2010] Baumeister, J. and Seipel, D. (2010). Anomalies in ontologies with rules. *J. Web Sem.*, 8(1):55–68.

[Baxter et al., 2006] Baxter, G., Frean, M. R., Noble, J., Rickerby, M., Smith, H., Visser, M., Melton, H., and Tempero, E. D. (2006). Understanding the shape of Java software. In *Proc. OOPSLA*, pages 397–412. ACM.

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

[Bézivin, 1998] Bézivin, J. (1998). Who's afraid of ontologies? In *Proc. OOPSLA*.

[Bézivin et al., 2005a] Bézivin, J., Brunette, C., Chevrel, R., Jouault, F., and Kurtev, I. (2005a). Bridging the generic modeling environment (gme) and the eclipse modeling framework (emf). In *Proc. OOPSLA*, volume 5.

[Bézivin et al., 2006] Bézivin, J., Favre, J., and Rumpe, B. (2006). First international workshop on global integrated model management. In *Proc. ICSE*, pages 1026–1027. ACM.

[Bézivin et al., 2006] Bézivin, J., Favre, J.-M., and Rumpe, B. (2006). Introduction to gamma 2006 first international workshop on global integrated model management. In *Proc. Global integrated model management.*

[Bézivin et al., 2005b] Bézivin, J., Hillairet, G., Jouault, F., Kurtev, I., and Piers, W. (2005b). Bridging the ms/dsl tools and the eclipse modeling framework. In *Proc. OOPSLA*, volume 5, pages 1–19.

[Bézivin et al., 2005c] Bézivin, J., Jouault, F., Rosenthal, P., and Valduriez, P. (2005c). Modeling in the Large and Modeling in the Small. In *Proc. MDAFA*, volume 3599 of *Lecture Notes in Computer Science*, pages 33–46. Springer.

[Bézivin et al., 2004] Bézivin, J., Jouault, F., and Valduriez, P. (2004). On the need for Megamodels. In *Proc. OOPSLA/GPCE*, pages 1–9.

[Biermann et al., 2010] Biermann, E., Ermel, C., and Jurack, S. (2010). Modeling the "ecore to genmodel" transformation with emf henshin. *Transformation Tool Contest*, page 153.

[Biswas et al., 2018] Biswas, R., Türker, R., Moghaddam, F. B., Koutraki, M., and Sack, H. (2018). Wikipedia infobox type prediction using embeddings. In *Proc. DL4KGS@ESWC*, volume 2106 of *CEUR Workshop Proceedings*, pages 46–55. CEUR-WS.org.

[Bizer et al., 2009] Bizer, C., Lehmann, J., Kobilarov, G., Auer, S., Becker, C., Cyganiak, R., and Hellmann, S. (2009). Dbpedia - A crystallization point for the web of data. *J. Web Sem.*, 7(3):154–165.

[Boinski et al., 2019] Boinski, T., Szymanski, J., and Cejrowski, T. (2019). Exact-match based wikipedia-wordnet integration. In *Proc. INISTA*, pages 1–6. IEEE.

[Bostandjiev et al., 2011] Bostandjiev, S., O'Donovan, J., Hall, C., Gretarsson, B., and Höllerer, T. (2011). Wigipedia: A tool for improving structured data in wikipedia. In *Proceedings of the 5th IEEE International Conference on Semantic Computing (ICSC 2011), Palo Alto, CA, USA, September 18-21, 2011*, pages 328–335. IEEE Computer Society.

[Bräuer and Lochmann, 2008] Bräuer, M. and Lochmann, H. (2008). An ontology for software models and its practical implications for semantic web reasoning. In *Proc. ESWC*, volume 5021 of *Lecture Notes in Computer Science*, pages 34–48. Springer.

[Brewster et al., 2004] Brewster, C., Alani, H., Dasmahapatra, S., and Wilks, Y. (2004). Data driven ontology evaluation. In *Proc. LREC*. European Language Resources Association.

[Buchmann and Schwägerl, 2015] Buchmann, T. and Schwägerl, F. (2015). On a-posteriori integration of ecore models and hand-written java code. In *Proc. ICSOFT-PT*, volume 2, pages 1–8. IEEE.

[Buffa et al., 2008] Buffa, M., Gandon, F. L., Erétéo, G., Sander, P., and Faron, C. (2008). Sweetwiki: A semantic wiki. *J. Web Semant.*, 6(1):84–97.

[Calero et al., 2006] Calero, C., Ruiz, F., and Piattini, M., editors (2006). *Ontologies for Software Engineering and Software Technology*. Springer.

[Cannaviccio et al., 2018] Cannaviccio, M., Ariemma, L., Barbosa, D., and Merialdo, P. (2018). Leveraging wikipedia table schemas for knowledge graph augmentation. In *Proc. Workshop on the Web and Databases*, pages 5:1–5:6. ACM.

[Ceri et al., 2013] Ceri, S., Palpanas, T., Valle, E. D., Pedreschi, D., Freytag, J., and Trasarti, R. (2013). Towards mega-modeling: a walk through data analysis experiences. *SIGMOD Record*, 42(3):19–27.

[Ceri et al., 2012] Ceri, S., Valle, E. D., Pedreschi, D., and Trasarti, R. (2012). Mega-modeling for big data analytics. In *Proc. ER*, volume 7532 of *Lecture Notes in Computer Science*, pages 1–15. Springer.

[Cesar Gonzalez-Perez, 2017] Cesar Gonzalez-Perez (2017). How Ontologies Can Help in Software Engineering. In *Proc. GTTSE*, volume 10223 of *Lecture Notes in Computer Science*, pages 26–44. Springer.

[Chawla et al., 2002] Chawla, N. V., Bowyer, K. W., Hall, L. O., and Kegelmeyer, W. P. (2002). SMOTE: synthetic minority over-sampling technique. *J. Artif. Intell. Res.*, 16:321–357.

[Cheatham and Hitzler, 2013] Cheatham, M. and Hitzler, P. (2013). String similarity metrics for ontology alignment. In *Proc. ISWC*, volume 8219 of *Lecture Notes in Computer Science*, pages 294–309. Springer.

[Chechik et al., 2016] Chechik, M., Famelis, M., Salay, R., and Strüber, D. (2016). Perspectives of model transformation reuse. In *Proc. IFM*, volume 9681 of *Lecture Notes in Computer Science*, pages 28–44. Springer.

[Chen et al., 2016] Chen, K., Dong, X., Zhu, J., and Shen, B. (2016). Building a domain knowledge base from wikipedia: a semi-supervised approach. In *Proc. SEKE*, pages 191–196. KSI Research Inc. and Knowledge Systems Institute Graduate School.

[Chevance and Heidet, 1978] Chevance, R. and Heidet, T. (1978). Static profile and dynamic behavior of cobol programs. *ACM SIGPLAN Notices*, 13(4):44–57.

[Chiticariu et al., 2018] Chiticariu, L., Danilevsky, M., Li, Y., Reiss, F., and Zhu, H. (2018). Systemt: Declarative text understanding for enterprise. In *Proc. NAACL-HLT*, pages 76–83. Association for Computational Linguistics.

[Chiticariu et al., 2013] Chiticariu, L., Li, Y., and Reiss, F. R. (2013). Rule-based information extraction is dead! long live rule-based information extraction systems! In *Proc. EMNLP*, pages 827–832. ACL.

[Combemale et al., 2017] Combemale, B., Lämmel, R., and Wyk, E. V. (2017). SLEBOK: the software language engineering body of knowledge (dagstuhl seminar 17342). *Dagstuhl Reports*.

[Combemale et al., 2018] Combemale, B., Lämmel, R., and Wyk, E. V. (2018). SLEBOK: The Software Language Engineering Body of Knowledge (DS 17342). *Dagstuhl Reports*.

[Corcho et al., 2006] Corcho, O., Fernández-López, M., and Gómez-Pérez, A. (2006). Ontological engineering: principles, methods, tools and languages. In *Ontologies for software engineering and software technology*. Springer.

[Cucerzan, 2007] Cucerzan, S. (2007). Large-scale named entity disambiguation based on wikipedia data. In *Proc. EMNLP-CoNLL*, pages 708–716. ACL.

[Dagenais and Robillard, 2012] Dagenais, B. and Robillard, M. P. (2012). Recovering traceability links between an API and its learning resources. In *Proc. ICSE*, pages 47–57. IEEE Computer Society.

[de Almeida Falbo et al., 2005] de Almeida Falbo, R., Ruy, F. B., and Moro, R. D. (2005). Using ontologies to add semantics to a software engineering environment. In *Proc. SEKE*, pages 151–156.

[de Souza and Bentolila, 2009] de Souza, C. R. B. and Bentolila, D. L. M. (2009). Automatic evaluation of API usability using complexity metrics and visualizations. In *Proc. ICSE*, pages 299–302. IEEE.

[Dellschaft and Staab, 2006] Dellschaft, K. and Staab, S. (2006). On how to perform a gold standard based evaluation of ontology learning. In *Proc. ISWC*, volume 4273 of *Lecture Notes in Computer Science*, pages 228–241. Springer.

[Diskin et al., 2013] Diskin, Z., Kokaly, S., and Maibaum, T. (2013). Mapping-Aware Megamodeling: Design Patterns and Laws. In *Proc. SLE*, volume 8225 of *Lecture Notes in Computer Science*, pages 322–343. Springer.

[Dittrich et al., 1995] Dittrich, K. R., Gatziu, S., and Geppert, A. (1995). The active database management system manifesto: A rulebase of ADBMS features. In *Proc. RIDS*, volume 985 of *Lecture Notes in Computer Science*, pages 3–20. Springer.

[Do and Roth, 2012] Do, Q. X. and Roth, D. (2012). Exploiting the wikipedia structure in local and global classification of taxonomic relations. *Natural Language Engineering*, 18(2):235–262.

[Dong et al., 2016] Dong, X., Chen, K., Zhu, J., and Shen, B. (2016). Learning to discover subsumptions between software engineering concepts in wikipedia. In *Proc. SEKE*, pages 147–152. KSI Research Inc. and Knowledge Systems Institute Graduate School.

[dos Santos et al., 2019] dos Santos, G. M., de Oliveira, T. C., Alencar, P. S. C., and Cowan, D. D. (2019). Retrieving curated stack overflow posts from project task similarities (S). In *Proc. SEKE*, pages 415–528.

[Duala-Ekoko and Robillard, 2012] Duala-Ekoko, E. and Robillard, M. P. (2012). Asking and answering questions about unfamiliar apis: An exploratory study. In *Proc. ICSE*, pages 266–276. IEEE Computer Society.

[Ed-Douibi et al., 2016] Ed-Douibi, H., Izquierdo, J. L. C., Gómez, A., Tisi, M., and Cabot, J. (2016). EMF-REST: generation of restful apis from models. In Ossowski, S., editor, *Proc. SAC*, pages 1446–1453. ACM.

[Eilertsen and Bagge, 2018] Eilertsen, A. M. and Bagge, A. H. (2018). Exploring api/client co-evolution. In *Proc. WAPI@ICSE*, pages 10–13. ACM.

[Elberzhager et al., 2012] Elberzhager, F., Münch, J., and Nha, V. T. N. (2012). A systematic mapping study on the combination of static and dynamic quality assurance techniques. *Information & Software Technology*, 54(1):1–15.

[Ellis et al., 2007] Ellis, B., Stylos, J., and Myers, B. A. (2007). The factory pattern in API design: A usability evaluation. In *Proc. ICSE*, pages 302–312. IEEE Computer Society.

[Fairchild et al., 2015] Fairchild, G., Silva, L. D., Valle, S. Y. D., and Segre, A. M. (2015). Eliciting disease data from wikipedia articles. *CoRR*, abs/1504.00657.

[Falleri et al., 2010] Falleri, J., Huchard, M., Lafourcade, M., Nebut, C., Prince, V., and Dao, M. (2010). Automatic extraction of a wordnet-like identifier network from software. In *Proc. ICPC*, pages 4–13. IEEE Computer Society.

[Famelis and Chechik, 2019] Famelis, M. and Chechik, M. (2019). Managing design-time uncertainty. *Software and System Modeling*, 18(2):1249–1284.

[Färber, 2019] Färber, M. (2019). The microsoft academic knowledge graph: A linked data source with 8 billion triples of scholarly data. In *Proc. ISWC*, volume 11779 of *Lecture Notes in Computer Science*, pages 113–129. Springer.

[Färber et al., 2018] Färber, M., Bartscherer, F., Menne, C., and Rettinger, A. (2018). Linked data quality of dbpedia, freebase, opencyc, wikidata, and YAGO. *Semantic Web*, 9(1):77–129.

[Favre, 2004] Favre, J. (2004). Cacophony: Metamodel-driven architecture recovery. In *Proc. WCRE*.

[Favre et al., 2011] Favre, J., Gasevic, D., Lämmel, R., and Pek, E. (2011). Empirical Language Analysis in Software Linguistics. In *Proc. SLE*, volume 6563 of *Lecture Notes in Computer Science*, pages 316–326. Springer.

[Favre et al., 2012a] Favre, J., Lämmel, R., Leinberger, M., Schmorleiz, T., and Varanovich, A. (2012a). Linking Documentation and Source Code in a Software Chrestomathy. In *Proc. WCRE*, pages 335–344. IEEE Computer Society.

[Favre et al., 2012b] Favre, J., Lämmel, R., Schmorleiz, T., and Varanovich, A. (2012b). 101companies: A Community Project on Software Technologies and Software Languages. In *Proc. TOOLS*, volume 7304 of *Lecture Notes in Computer Science*, pages 58–74. Springer.

[Favre et al., 2012c] Favre, J., Lämmel, R., and Varanovich, A. (2012c). Modeling the Linguistic Architecture of Software Products. In *Proc. MODELS*, volume 7590 of *Lecture Notes in Computer Science*, pages 151–167. Springer.

[Favre, 2005a] Favre, J.-M. (2005a). Foundations of meta-pyramids: Languages vs. metamodels – Episode II: Story of thotus the baboon. In *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings.

[Favre, 2005b] Favre, J.-M. (2005b). Foundations of Model (Driven) (Reverse) Engineering: Models – Episode I: Stories of The Fidus Papyrus and of The Solarus. In *Language Engineering for Model-Driven Software Development*, Dagstuhl Seminar Proceedings.

[Favre, 2005c] Favre, J.-M. (2005c). Megamodeling and etymology-a story of words: From med to mde via model in five milleniums. In *In Dagstuhl Seminar on Transformation Techniques in Software Engineering*.

[Favre et al., 2009] Favre, J.-M., Gasevic, D., Lämmel, R., and Winter, A. (2009). Guest editors' introduction to the special section on software language engineering. *IEEE Trans. Software Eng.*, 35(6):737–741.

[Favre and Martinez, 2006] Favre, L. and Martinez, L. (2006). Formalizing MDA components. In *Proc. ICSR*, volume 4039 of *Lecture Notes in Computer Science*, pages 326–339. Springer.

[Feilkas and Ratiu, 2008] Feilkas, M. and Ratiu, D. (2008). Ensuring well-behaved usage of apis through syntactic constraints. In *Proc. ICPC*, pages 248–253. IEEE Computer Society.

[Fernández et al., 2019] Fernández, D. M., Böhm, W., Vogelsang, A., Mund, J., Broy, M., Kuhrmann, M., and Weyer, T. (2019). Artefacts in software engineering: a fundamental positioning. *Software and System Modeling*, 18(5):2777–2786.

[Fetahu et al., 2019] Fetahu, B., Anand, A., and Koutraki, M. (2019). Tablenet: An approach for determining fine-grained relations for wikipedia tables. In *Proc. WWW*, pages 2736–2742. ACM.

[Flati et al., 2014] Flati, T., Vannella, D., Pasini, T., and Navigli, R. (2014). Two Is Bigger (and Better) Than One: the Wikipedia Bitaxonomy Project. In *Proc. ACL*, pages 945–955. The Association for Computer Linguistics.

[Flati et al., 2016] Flati, T., Vannella, D., Pasini, T., and Navigli, R. (2016). Multiwibi: The multilingual wikipedia bitaxonomy project. *Artif. Intell.*, 241:66–102.

[Floyd, 1979] Floyd, R. W. (1979). The Paradigms of Programming. *Commun. ACM*, 22(8):455–460.

[Forward and Lethbridge, 2008] Forward, A. and Lethbridge, T. C. (2008). A taxonomy of software types to facilitate search and evidence-based software engineering. In *Proc. Centre for Advanced Studies on Collaborative Research*. IBM.

[Fowler, 2010] Fowler, M. (2010). *Domain-specific languages*. Pearson Education.

[Frikh et al., 2011] Frikh, B., Djaanfar, A. S., and Ouhbi, B. (2011). A hybrid method for domain ontology construction from the web. In *Proc. KEOD*, pages 285–292. SciTePress.

[Fritzsche et al., 2008] Fritzsche, M., Johannes, J., Aßmann, U., Mitschke, S., Gilani, W., Spence, I. T. A., Brown, T. J., and Kilpatrick, P. (2008). Systematic usage of embedded modelling languages in automated model transformation chains. In *Proc. SLE*, volume 5452 of *Lecture Notes in Computer Science*, pages 134–150. Springer.

[Gangemi et al., 2004] Gangemi, A., Borgo, S., Catenacci, C., and Lehmann, J. (2004). Task taxonomies for knowledge content. *Metokis Deliverable D*, 7:2004.

[Gangemi et al., 2002] Gangemi, A., Guarino, N., Masolo, C., Oltramari, A., and Schneider, L. (2002). Sweetening Ontologies with DOLCE. In *Proc. EKAW*, volume 2473 of *Lecture Notes in Computer Science*, pages 166–181. Springer.

[Gangemi and Mika, 2003] Gangemi, A. and Mika, P. (2003). Understanding the semantic web through descriptions and situations. In *Proc. OTM*, volume 2888 of *Lecture Notes in Computer Science*, pages 689–706. Springer.

[Garmendia et al., 2014] Garmendia, A., Guerra, E., Kolovos, D. S., and de Lara, J. (2014). EMF Splitter: A Structured Approach to EMF Modularity. In *XM@MoDELS*, volume 1239 of *CEUR Workshop Proceedings*, pages 22–31. CEUR-WS.org.

[Gascueña et al., 2012] Gascueña, J. M., Navarro, E., and Fernández-Caballero, A. (2012). Model-driven engineering techniques for the development of multi-agent systems. *Eng. Appl. of AI*, 25(1):159–173.

[Gasevic et al., 2007] Gasevic, D., Kaviani, N., and Hatala, M. (2007). On metamodeling in megamodels. In *Proc. MoDELS*, volume 4735 of *Lecture Notes in Computer Science*, pages 91–105. Springer.

[Gerbig et al., 2016] Gerbig, R., Atkinson, C., de Lara, J., and Guerra, E. (2016). A feature-based comparison of melanee and metadepth. In *Proc. MoDELS*, volume 1722 of *CEUR Workshop Proceedings*, pages 25–34. CEUR-WS.org.

[Giles, 2005] Giles, J. (2005). Internet encyclopaedias go head to head.

[Gruber, 1995] Gruber, T. R. (1995). Toward principles for the design of ontologies used for knowledge sharing? *Int. J. Hum. Comput. Stud.*, 43(5-6):907–928.

[Guizzardi, 2009] Guizzardi, G. (2009). The problem of transitivity of part-whole relations in conceptual modeling revisited. In *Proc. CAiSE*, volume 5565 of *Lecture Notes in Computer Science*, pages 94–109. Springer.

[Gupta and Manning, 2014] Gupta, S. and Manning, C. D. (2014). Improved pattern learning for bootstrapped entity extraction. In *Computational Natural Language Learning (CoNLL)*, pages 98–108. ACL.

[Hahsler, 2003] Hahsler, M. (2003). A quantitative study of the application of design patterns in java. Working Papers on Information Systems, Information Business and Operations 01/2003, Vienna.

[Härtel et al., 2018a] Härtel, J., Aksu, H., and Lämmel, R. (2018a). Classification of apis by hierarchical clustering. In *Proc. ICPC*, pages 233–243. ACM.

[Härtel et al., 2017] Härtel, J., Härtel, L., Lämmel, R., Varanovich, A., and Heinz, M. (2017). Interconnected linguistic architecture. *Art Sci. Eng. Program.*, 1(1):3.

[Härtel et al., 2018b] Härtel, J., Heinz, M., and Lämmel, R. (2018b). EMF patterns of usage on github. In *Proc. ECMFA*, volume 10890 of *Lecture Notes in Computer Science*, pages 216–234. Springer.

[Härtel and Lämmel, 2020] Härtel, J. and Lämmel, R. (2020). Incremental map-reduce on repository history. In *SANER*, pages 320–331. IEEE.

[Hauff and Gousios, 2015] Hauff, C. and Gousios, G. (2015). Matching github developer profiles to job advertisements. In *Proc. MSR*, pages 362–366. IEEE Computer Society.

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

[Hebig et al., 2012] Hebig, R., Gabrysiak, G., and Giese, H. (2012). Towards patterns for mde-related processes to detect and handle changeability risks. In *Proc. ICSSP*, pages 38–47. IEEE Computer Society.

[Hebig and Giese, 2017] Hebig, R. and Giese, H. (2017). On the complex nature of MDE evolution and its impact on changeability. *Software & Systems Modeling*, 16(2):333–356.

[Heinrich, 2016] Heinrich, R. (2016). Architectural run-time models for performance and privacy analysis in dynamic cloud applications? *SIGMETRICS Performance Evaluation Review*, 43(4):13–22.

[Heinz et al., 2020] Heinz, M., Härtel, J., and Lämmel, R. (2020). Reproducible construction of interconnected technology models for EMF code generation. *J. Object Technol.*, 19(2):8:1–25.

[Heinz et al., 2016] Heinz, M., Helsper, P., Lämmel, R., and Schmidt, T. M. (2016). A DSL for executable 'how to' manuals. In *Proc. SAC*, pages 2007–2009. ACM.

[Heinz et al., 2019] Heinz, M., Lämmel, R., and Acher, M. (2019). Discovering indicators for classifying wikipedia articles in a domain - A case study on software languages. In *Proc. SEKE*, pages 541–706. KSI.

[Heinz et al., 2017] Heinz, M., Lämmel, R., and Varanovich, A. (2017). Axioms of linguistic architecture. In *Proc. MODELSWARD*, pages 478–486. SciTePress.

[Hilliard et al., 2010] Hilliard, R., Malavolta, I., Muccini, H., and Pelliccione, P. (2010). Realizing architecture frameworks through megamodelling techniques. In *Proc. ASE*, pages 305–308. ACM.

[Hlomani and Stacey, 2014] Hlomani, H. and Stacey, D. A. (2014). Multiple dimensions to data-driven ontology evaluation. In *Proc. IC3K*, volume 553 of *Communications in Computer and Information Science*, pages 329–346. Springer.

[Hogan et al., 2020] Hogan, A., Blomqvist, E., Cochez, M., d'Amato, C., de Melo, G., Gutierrez, C., Gayo, J. E. L., Kirrane, S., Neumaier, S., Polleres, A., Navigli, R., Ngomo, A.-C. N., Rashid, S. M., Rula, A., Schmelzeisen, L., Sequeda, J., Staab, S., and Zimmermann, A. (2020). Knowledge graphs.

[Hu et al., 2009] Hu, X., Zhang, X., Lu, C., Park, E. K., and Zhou, X. (2009). Exploiting wikipedia as external knowledge for document clustering. In *Proc. SIGKDD*, pages 389–396. ACM.

[Huang et al., 2020] Huang, C., Yao, L., Wang, X., Benatallah, B., and Zhang, X. (2020). Software expert discovery via knowledge domain embeddings in a collaborative network. *Pattern Recognit. Lett.*, 130:46–53.

[Jouault et al., 2010] Jouault, F., Vanhooff, B., Brunelière, H., Doux, G., Berbers, Y., and Bézivin, J. (2010). Inter-dsl coordination support by combining megamodeling and model weaving. In *Proc. SAC*, pages 2011–2018. ACM.

[Kahani et al., 2016] Kahani, N., Bagherzadeh, M., Dingel, J., and Cordy, J. R. (2016). The problems with eclipse modeling tools: A topic analysis of Eclipse forums. In *Proc. of MODELS*, pages 227–237. ACM.

[Kahanwal, 2013] Kahanwal, B. (2013). Abstraction level taxonomy of programming language frameworks. *CoRR*, abs/1311.3293.

[Kaptein et al., 2010] Kaptein, R., Serdyukov, P., de Vries, A. P., and Kamps, J. (2010). Entity ranking using wikipedia as a pivot. In *Proc. CIKM*, pages 69–78. ACM.

[Karus and Gall, 2011] Karus, S. and Gall, H. C. (2011). A study of language usage evolution in open source software. In *Proc. MSR*, pages 13–22. ACM.

[Khatchadourian et al., 2020] Khatchadourian, R., Tang, Y., Bagherzadeh, M., and Ray, B. (2020). An empirical study on the use and misuse of java 8 streams. In *Proc. FASE*, volume 12076 of *Lecture Notes in Computer Science*, pages 97–118. Springer.

[Kitchenham et al., 2009] Kitchenham, B. A., Brereton, P., Budgen, D., Turner, M., Bailey, J., and Linkman, S. G. (2009). Systematic literature reviews in software engineering - A systematic literature review. *Information & Software Technology*, 51(1):7–15.

[Kitchenham et al., 1999] Kitchenham, B. A., Travassos, G. H., von Mayrhauser, A., Niessink, F., Schneidewind, N. F., Singer, J., Takada, S., Vehvilainen, R., and Yang, H. (1999). Towards an ontology of software maintenance. *Journal of Software Maintenance*, 11(6):365–389.

[Kleppe, 2008] Kleppe, A. (2008). *Software language engineering: creating domain-specific languages using metamodels*. Pearson Education.

[Kling et al., 2011] Kling, W., Jouault, F., Wagelaar, D., Brambilla, M., and Cabot, J. (2011). Moscript: A DSL for querying and manipulating model repositories. In *Proc. SLE*, volume 6940 of *Lecture Notes in Computer Science*, pages 180–200. Springer.

[Kolovos et al., 2017] Kolovos, D. S., García-Domínguez, A., Rose, L. M., and Paige, R. F. (2017). Eugenia: towards disciplined and automated development of gmf-based graphical model editors. *Software and Systems Modeling*, 16(1):229–255.

[Kolovos et al., 2015] Kolovos, D. S., Matragkas, N. D., Korkontzelos, I., Ananiadou, S., and Paige, R. F. (2015). Assessing the use of Eclipse MDE technologies in open-source software projects. In *Proc. MODELS*, volume 1541 of *CEUR Workshop Proceedings*, pages 20–29. CEUR-WS.org.

[Kolovos et al., 2010] Kolovos, D. S., Rose, L. M., bin Abid, S., Paige, R. F., Polack, F. A. C., and Botterweck, G. (2010). Taming EMF and GMF Using Model Transformation. In *Proc. MODELS*, volume 6394 of *Lecture Notes in Computer Science*, pages 211–225. Springer.

[Kuhn et al., 2016] Kuhn, P., Mischkewitz, S., Ring, N., and Windheuser, F. (2016). Type inference on wikipedia list pages. In *46. Jahrestagung der Gesellschaft für Informatik*, volume P-259 of *LNI*, pages 2101–2111. GI.

[Kühne, 2016] Kühne, T. (2016). Unifying explanatory and constructive modeling: towards removing the gulf between ontologies and conceptual models. In *Proc. MoDELS*, pages 95–102. ACM.

[Kühne, 2018] Kühne, T. (2018). Exploring potency. In *Proc. MODELS*, pages 2–12. ACM.

[Kumar et al., 2017] Kumar, S., Baliyan, N., and Sukalikar, S. (2017). Ontology cohesion and coupling metrics. *Int. J. Semantic Web Inf. Syst.*, 13(4):1–26.

[Kurtev et al., 2002] Kurtev, I., Bézivin, J., and Akşit, M. (2002). Technological Spaces: an Initial Appraisal. In *Proc. CoopIS, DOA*.

[Kusel et al., 2013] Kusel, A., Schoenboeck, J., Wimmer, M., Retschitzegger, W., Schwinger, W., and Kappel, G. (2013). Reality check for model transformation reuse: The ATL transformation zoo case study. In *Proc. AMT*, volume 1077 of *CEUR Workshop Proceedings*. CEUR-WS.org.

[Lämmel, 2015] Lämmel, R. (2015). Software chrestomathies. *Sci. Comput. Program.*, 97:98–104.

[Lämmel, 2016] Lämmel, R. (2016). Coupled software transformations revisited. In *Proc. SLE*, pages 239–252. ACM.

[Lämmel, 2016] Lämmel, R. (2016). Coupled Software Transformations Revisited. In *Proc. SLE*, pages 239–252. ACM.

[Lämmel, 2017] Lämmel, R. (2017). Relationship maintenance in software language repositories. *Art Sci. Eng. Program.*, 1(1):4.

[Lämmel, 2018] Lämmel, R. (2018). *Software languages: Syntax, semantics, and metaprogramming*. Springer.

[Lämmel et al., 2014a] Lämmel, R., Leinberger, M., Schmorleiz, T., and Varanovich, A. (2014a). Comparison of feature implementations across languages, technologies, and styles. In *Proc. CSMR-WCRE*, pages 333–337. IEEE Computer Society.

[Lämmel and Meijer, 2007] Lämmel, R. and Meijer, E. (2007). Revealing the X/O Impedance Mismatch - (Changing Lead into Gold). In *Proc. SSDGP*, volume 4719 of *Lecture Notes in Computer Science*, pages 285–367. Springer.

[Lämmel et al., 2013a] Lämmel, R., Mosen, D., and Varanovich, A. (2013a). Method and Tool Support for Classifying Software Languages with Wikipedia. In *Proc. SLE*, volume 8225 of *Lecture Notes in Computer Science*, pages 249–259. Springer.

[Lämmel et al., 2013b] Lämmel, R., Mosen, D., and Varanovich, A. (2013b). Method and Tool Support for Classifying Software Languages with Wikipedia. In *Proc. SLE*, volume 8225 of *Lecture Notes in Computer Science*, pages 249–259. Springer.

[Lämmel and Pek, 2013] Lämmel, R. and Pek, E. (2013). Understanding privacy policies - A study in empirical analysis of language usage. *Empirical Software Engineering*, 18(2):310–374.

[Lämmel et al., 2013c] Lämmel, R., Schmorleiz, T., and Varanovich, A. (2013c). The 101haskell Chrestomathy: A Whole Bunch of Learnable Lambdas. In *Proc. IFL*, page 25. ACM.

[Lämmel and Varanovich, 2014] Lämmel, R. and Varanovich, A. (2014). Interpretation of Linguistic Architecture. In *Proc. ECMFA*, volume 8569 of *Lecture Notes in Computer Science*, pages 67–82. Springer.

[Lämmel et al., 2014b] Lämmel, R., Varanovich, A., Leinberger, M., Schmorleiz, T., and Favre, J. (2014b). Declarative Software Development: Distilled Tutorial. In *Proc. PPDP*, pages 1–6. ACM.

[Lämmel and Zaytsev, 2013] Lämmel, R. and Zaytsev, V. (2013). Language support for megamodel renarration. In *Proc. Extreme Modeling@MODELS*, volume 1089 of *CEUR Workshop Proceedings*, pages 36–45. CEUR-WS.org.

[Landis and Koch, 1977] Landis, J. R. and Koch, G. G. (1977). The measurement of observer agreement for categorical data. *biometrics*, pages 159–174.

[Lange and Atkinson, 2018] Lange, A. and Atkinson, C. (2018). Multi-level modeling with MELANEE. In *Proc. MODELS*, volume 2245 of *CEUR Workshop Proceedings*, pages 653–662. CEUR-WS.org.

[Langer et al., 2011] Langer, P., Wieland, K., Wimmer, M., and Cabot, J. (2011). From UML profiles to EMF profiles and beyond. In *Proc. TOOLS*, volume 6705 of *Lecture Notes in Computer Science*, pages 52–67. Springer.

[Langer et al., 2012] Langer, P., Wieland, K., Wimmer, M., and Cabot, J. (2012). EMF profiles: A lightweight extension approach for EMF models. *Journal of Object Technology*, 11(1):1–29.

[Li et al., 2018] Li, J., Sun, A., and Xing, Z. (2018). To do or not to do: Distill crowdsourced negative caveats to augment api documentation. *J. Assoc. Inf. Sci. Technol.*, 69(12):1460–1475.

[Lima et al., 2016] Lima, L. G., Soares-Neto, F., Lieuthier, P., Castor, F., Melfe, G., and Fernandes, J. P. (2016). Haskell in green land: Analyzing the energy behavior of a purely functional language. In *Proc. SANER*, pages 517–528. IEEE Computer Society.

[Lockard et al., 2018] Lockard, C., Dong, X. L., Shiralkar, P., and Einolghozati, A. (2018). CERES: distantly supervised relation extraction from the semi-structured web. *Proc. VLDB Endow.*, 11(10):1084–1096.

[Lount and Bunt, 2014] Lount, M. and Bunt, A. (2014). Characterizing web-based tutorials: Exploring quality, community, and showcasing strategies. In *Proc. SIGDOC*, pages 6:1–6:10. ACM.

[Lu et al., 2013] Lu, C., Bing, L., Lam, W., Chan, K., and Gu, Y. (2013). Web entity detection for semi-structured text data records with unlabeled data. *International Journal of Computational Linguistics and Applications*, 4(2):135–150.

[Lu and Du, 2017] Lu, Q. and Du, Y. (2017). Wikipedia-based entity semantifying in open information extraction. In *Proc. ICDAR*, pages 765–770. IEEE.

[Lv et al., 2015] Lv, F., Zhang, H., Lou, J., Wang, S., Zhang, D., and Zhao, J. (2015). Codehow: Effective code search based on API understanding and extended boolean model (E). In *Proc. ASE*, pages 260–270. IEEE Computer Society.

[Lyu et al., 2017] Lyu, Y., Gui, J., Wan, M., and Halfond, W. G. J. (2017). An Empirical Study of Local Database Usage in Android Applications. In *Proc. ICSME*, pages 444–455. IEEE.

[Maalej and Robillard, 2013] Maalej, W. and Robillard, M. P. (2013). Patterns of knowledge in API reference documentation. *IEEE Trans. Software Eng.*, 39(9):1264–1282.

[Maedche and Staab, 2002] Maedche, A. and Staab, S. (2002). Measuring similarity between ontologies. In *Proc. EKAW*, volume 2473 of *Lecture Notes in Computer Science*, pages 251–263. Springer.

[Malloy and Power, 2019] Malloy, B. A. and Power, J. F. (2019). An empirical analysis of the transition from python 2 to python 3. *Empirical Software Engineering*, 24(2):751–778.

[Malone et al., 2014] Malone, J., Brown, A., Lister, A. L., Ison, J., Hull, D., Parkinson, H., and Stevens, R. (2014). The software ontology (swo): a resource for reproducibility in biomedical data analysis, curation and digital preservation. *Journal of biomedical semantics*, 5:25.

[Manning et al., 2014] Manning, C. D., Surdeanu, M., Bauer, J., Finkel, J., Bethard, S. J., and McClosky, D. (2014). The Stanford CoreNLP natural language processing toolkit. In *(ACL) System Demonstrations*, pages 55–60. The Association for Computer Linguistics.

[Martínez-Rodríguez et al., 2020] Martínez-Rodríguez, J., Hogan, A., and López-Arévalo, I. (2020). Information extraction meets the semantic web: A survey. *Semantic Web*, 11(2):255–335.

[Mayer and Bauer, 2015] Mayer, P. and Bauer, A. (2015). An empirical analysis of the utilization of multiple programming languages in open source projects. In *Proc. EASE*, pages 4:1–4:10. ACM.

[Medvidovic and Taylor, 2000] Medvidovic, N. and Taylor, R. N. (2000). A classification and comparison framework for software architecture description languages. *IEEE Trans. Software Eng.*, 26(1):70–93.

[Mendes et al., 2011] Mendes, P. N., Jakob, M., García-Silva, A., and Bizer, C. (2011). Dbpedia spotlight: shedding light on the web of documents. In *Proc. I-SEMANTICS*, ACM International Conference Proceeding Series, pages 1–8. ACM.

[Méndez-Acuña et al., 2013] Méndez-Acuña, D., Casallas, R., and Etien, A. (2013). On the customization of model management systems for file-centric ides. In *Proc. DSM@SPLASH*, pages 57–62. ACM.

[Metke-Jimenez et al., 2010] Metke-Jimenez, A., Raymond, K., and MacColl, I. (2010). Ontologies derived from wikipedia - A framework for comparison. In *Proc. KEOD*, pages 382–387. SciTePress.

[Meyerovich and Rabkin, 2013] Meyerovich, L. A. and Rabkin, A. S. (2013). Empirical analysis of programming language adoption. In *Proc. OOPSLA*, pages 1–18. ACM.

[Miller and Fellbaum, 2007] Miller, G. A. and Fellbaum, C. (2007). Wordnet then and now. *Lang. Resour. Evaluation*, 41(2):209–214.

[Mintz et al., 2009] Mintz, M., Bills, S., Snow, R., and Jurafsky, D. (2009). Distant supervision for relation extraction without labeled data. In *Proc. ACL*, pages 1003–1011. The Association for Computer Linguistics.

[Mirylenka et al., 2015] Mirylenka, D., Passerini, A., and Serafini, L. (2015). Bootstrapping domain ontologies from wikipedia: A uniform approach. In *Proc. IJCAI*, pages 1464–1470. AAAI Press.

[Morales et al., 2016] Morales, A., Premtoon, V., Avery, C., Felshin, S., and Katz, B. (2016). Learning to answer questions from wikipedia infoboxes. In *Proc. EMNLP*, pages 1930–1935. The Association for Computational Linguistics.

[Morsey et al., 2012] Morsey, M., Lehmann, J., Auer, S., Stadler, C., and Hellmann, S. (2012). Dbpedia and the live extraction of structured data from wikipedia. *Program*, 46(2):157–181.

[Nassif et al., 2019] Nassif, M., Treude, C., and Robillard, M. P. (2019). Witt: querying technology terms based on automated classification. In *Proc. ICSE*, pages 63–66. IEEE / ACM.

[Nassif et al., 2020] Nassif, M., Treude, C., and Robillard, M. P. (2020). Automatically categorizing software technologies. *IEEE Trans. Software Eng.*, 46(1):20–32.

[Nastase and Strube, 2008] Nastase, V. and Strube, M. (2008). Decoding Wikipedia categories for knowledge acquisition. In *Proc. AAAI*, pages 1219–1224. AAAI Press.

[Navarro et al., 2010] Navarro, J. F. G., García-Peñalvo, F. J., and Therón, R. (2010). A survey on ontology metrics. In *Proc. WSKS*, volume 111 of *Communications in Computer and Information Science*, pages 22–27. Springer.

[Navigli and Ponzetto, 2012] Navigli, R. and Ponzetto, S. P. (2012). Babelnet: The automatic construction, evaluation and application of a wide-coverage multilingual semantic network. *Artif. Intell.*, 193:217–250.

[Nguyen et al., 2007] Nguyen, D. P. T., Matsuo, Y., and Ishizuka, M. (2007). Relation extraction from wikipedia using subtree mining. In *Proc. AAAI*, pages 1414–1420. AAAI Press.

[Niu et al., 2017] Niu, H., Keivanloo, I., and Zou, Y. (2017). API usage pattern recommendation for software development. *J. Syst. Softw.*, 129:127–139.

[Novak et al., 2010] Novak, J., Krajnc, A., et al. (2010). Taxonomy of static code analysis tools. In *Proc. MIPRO*, pages 418–422. IEEE.

[Nuzzolese et al., 2011] Nuzzolese, A. G., Gangemi, A., Presutti, V., and Ciancarini, P. (2011). Encyclopedic Knowledge Patterns from Wikipedia Links. In *Proc. ISWC*, volume 7031 of *Lecture Notes in Computer Science*, pages 520–536. Springer.

[Nykaza et al., 2002] Nykaza, J., Messinger, R., Boehme, F., Norman, C. L., Mace, M., and Gordon, M. (2002). What programmers really want: results of a needs assessment for SDK documentation. In *Proc. SIGDOC*, pages 133–141. ACM.

[Oberle et al., 2004] Oberle, D., Eberhart, A., Staab, S., and Volz, R. (2004). Developing and Managing Software Components in an Ontology-Based Application Server. In *Proc. Middleware*, volume 3231 of *Lecture Notes in Computer Science*, pages 459–477. Springer.

[Oberle et al., 2009] Oberle, D., Grimm, S., and Staab, S. (2009). An ontology for software. In *Handbook on Ontologies*. Springer.

[Oberle et al., 2006] Oberle, D., Lamparter, S., Grimm, S., Vrandecic, D., Staab, S., and Gangemi, A. (2006). Towards ontologies for formalizing modularization and communication in large software systems. *Applied Ontology*, 1(2):163–202.

[Parnin and Orso, 2011] Parnin, C. and Orso, A. (2011). Are automated debugging techniques actually helping programmers? In *Proc. ISSTA*, pages 199–209.

[Parra et al., 2018] Parra, E., Escobar-Avila, J., and Haiduc, S. (2018). Automatic tag recommendation for software development video tutorials. In *Proc. ICPC*, pages 222–232. ACM.

[Perovich et al., 2009] Perovich, D., Bastarrica, M. C., and Rojas, C. (2009). Model-driven approach to software architecture design. In *Proc. SHARK@ICSE*, pages 1–8. IEEE Computer Society.

[Pescador et al., 2015] Pescador, A., Garmendia, A., Guerra, E., Cuadrado, J. S., and de Lara, J. (2015). Pattern-based development of domain-specific modelling languages. In *Proc. MoDELS*, pages 166–175. IEEE Computer Society.

[Petersen et al., 2008] Petersen, K., Feldt, R., Mujtaba, S., and Mattsson, M. (2008). Systematic Mapping Studies in Software Engineering. In *Proc. EASE*, Workshops in Computing. BCS.

[Petrosyan et al., 2015] Petrosyan, G., Robillard, M. P., and Mori, R. D. (2015). Discovering information explaining API types using text classification. In *Proc. ICSE*, pages 869–879. IEEE Computer Society.

[Ponzanelli et al., 2016] Ponzanelli, L., Bavota, G., Mocci, A., Penta, M. D., Oliveto, R., Hasan, M. A., Russo, B., Haiduc, S., and Lanza, M. (2016). Too long; didn't watch!: extracting relevant fragments from software development video tutorials. In *Proc. ICSE*, pages 261–272. ACM.

[Powers, 2003] Powers, D. M. (2003). Recall & precision versus the bookmaker. In *Proc. International Conference on Cognitive Science*.

[Powers, 2012] Powers, D. M. W. (2012). The problem with kappa. In *Proc. EACL*, pages 345–355. The Association for Computer Linguistics.

[Prana et al., 2019] Prana, G. A. A., Treude, C., Thung, F., Atapattu, T., and Lo, D. (2019). Categorizing the content of github readme files. *Empirical Software Engineering*, 24(3):1296–1327.

[Presutti et al., 2014] Presutti, V., Consoli, S., Nuzzolese, A. G., Recupero, D. R., Gangemi, A., Bannour, I., and Zargayouna, H. (2014). Uncovering the Semantics of Wikipedia Pagelinks. In *Proc. EKAW*, volume 8876 of *Lecture Notes in Computer Science*, pages 413–428. Springer.

[Qi et al., 2010] Qi, Y.-d., Qu, N., and Xie, X.-f. (2010). Towards a preliminary ontology for conceptual model quality evaluating. In *Proc. Web Information Systems and Mining*, volume 1, pages 329–334. IEEE.

[Raad and Cruz, 2015] Raad, J. and Cruz, C. (2015). A Survey on Ontology Evaluation Methods. In *Proc. KEOD*, pages 179–186. SciTePress.

[Ralf Lämmel and Stan Kitsis and Dave Remy, 2005] Ralf Lämmel and Stan Kitsis and Dave Remy (2005). Analysis of XML schema usage. In *Proc. XML*, volume 5. Citeseer.

[Ratiu and Deissenboeck, 2006] Ratiu, D. and Deissenboeck, F. (2006). Programs are knowledge bases. In *Proc. ICPC*, pages 79–83. IEEE Computer Society.

[Ratiu et al., 2008] Ratiu, D., Feilkas, M., and Jürjens, J. (2008). Extracting domain ontologies from domain specific apis. In *Proc. CSMR*, pages 203–212. IEEE Computer Society.

[Ratiu and Jürjens, 2007] Ratiu, D. and Jürjens, J. (2007). The reality of libraries. In *Proc. CSMR*, pages 307–318. IEEE Computer Society.

[Ratiu and Jürjens, 2008] Ratiu, D. and Jürjens, J. (2008). Evaluating the reference and representation of domain concepts in apis. In *Proc. ICPC*, pages 242–247. IEEE Computer Society.

[Rebele et al., 2016] Rebele, T., Suchanek, F. M., Hoffart, J., Biega, J., Kuzey, E., and Weikum, G. (2016). YAGO: A multilingual knowledge base from wikipedia, wordnet, and geonames. In *Proc. ISWC*, volume 9982 of *Lecture Notes in Computer Science*, pages 177–185.

[Rebouças et al., 2016] Rebouças, M., Pinto, G., Ebert, F., Torres, W., Serebrenik, A., and Castor, F. (2016). An empirical study on the usage of the swift programming language. In *Proc. SANER*, pages 634–638. IEEE Computer Society.

[Ren et al., 2014] Ren, Y., Parvizi, A., Mellish, C., Pan, J. Z., van Deemter, K., and Stevens, R. (2014). Towards competency question-driven ontology authoring. In *Proc. ESWC*, volume 8465 of *Lecture Notes in Computer Science*, pages 752–767. Springer.

[Riedel et al., 2010] Riedel, S., Yao, L., and McCallum, A. (2010). Modeling relations and their mentions without labeled text. In *Proc. ECML PKDD*, volume 6323 of *Lecture Notes in Computer Science*, pages 148–163. Springer.

[Rios-Alvarado et al., 2011] Rios-Alvarado, A. B., López-Arévalo, I., and Sosa-Sosa, V. (2011). Structuring taxonomies by using linguistic patterns and wordnet on web search. In *Proc. KEOD*, pages 273–278. SciTePress.

[Rivera, 2010] Rivera, J. E. (2010). *On the semantics of real-time domain specific modeling languages*. PhD thesis, Ph. D. thesis, Universidad de Málaga.

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

[Robillard et al., 2013] Robillard, M. P., Bodden, E., Kawrykow, D., Mezini, M., and Ratchford, T. (2013). Automated API property inference techniques. *IEEE Trans. Software Eng.*, 39(5):613–637.

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

[Robillard and Treude, 2020] Robillard, M. P. and Treude, C. (2020). Understanding wikipedia as a resource for opportunistic learning of computing concepts. In *Proc. SIGCSE*, pages 72–78. ACM.

[Robles et al., 2017] Robles, G., Ho-Quang, T., Hebig, R., Chaudron, M. R. V., and Fernández, M. A. (2017). An extensive dataset of UML models in GitHub. In *Proc. MSR*, pages 519–522. IEEE Computer Society.

[Rocco et al., 2018] Rocco, J. D., Ruscio, D. D., Härtel, J., Iovino, L., Lämmel, R., and Pierantonio, A. (2018). Systematic recovery of MDE technology usage. In *Proc. STAF*, volume 10888 of *Lecture Notes in Computer Science*, pages 110–126. Springer.

[Rocco et al., 2020] Rocco, J. D., Ruscio, D. D., Härtel, J., Iovino, L., Lämmel, R., and Pierantonio, A. (2020). Understanding MDE projects: megamodels to the rescue for architecture recovery. *Software and Systems Modeling*, 19(2):401–423.

[Rocco et al., 2017] Rocco, J. D., Ruscio, D. D., Heinz, M., Iovino, L., Lämmel, R., and Pierantonio, A. (2017). Consistency recovery in interactive modeling. In *Proc. MODELS*, volume 2019 of *CEUR Workshop Proceedings*, pages 116–122. CEUR-WS.org.

[Roller et al., 2018] Roller, S., Kiela, D., and Nickel, M. (2018). Hearst patterns revisited: Automatic hypernym detection from large text corpora. In *Proc. ACL*, pages 358–363. Association for Computational Linguistics.

[Roover, 2011] Roover, C. D. (2011). A logic meta-programming foundation for example-driven pattern detection in object-oriented programs. In *Proc. ICSM*, pages 556–561. IEEE Computer Society.

[Roover et al., 2013] Roover, C. D., Lämmel, R., and Pek, E. (2013). Multi-dimensional exploration of API usage. In *Proc. ICPC*, pages 152–161. IEEE Computer Society.

[Ruiz et al., 2004] Ruiz, F., Barceló, A. V., Piattini, M., and García, F. (2004). An ontology for the management of software maintenance projects. *International Journal of Software Engineering and Knowledge Engineering*, 14(3):323–349.

[Saied et al., 2015] Saied, M. A., Benomar, O., Abdeen, H., and Sahraoui, H. A. (2015). Mining Multi-level API Usage Patterns. In *Proc. SANER*, pages 23–32. IEEE Computer Society.

[Saied et al., 2018] Saied, M. A., Ouni, A., Sahraoui, H. A., Kula, R. G., Inoue, K., and Lo, D. (2018). Improving reusability of software libraries through usage pattern mining. *J. Syst. Softw.*, 145:164–179.

[Saile, 2010] Saile, D. (2010). Integrating twouse and ocl-dl. Student project. University of Koblenz-Landau.

[Salay et al., 2015] Salay, R., Kokaly, S., Sandro, A. D., and Chechik, M. (2015). Enriching megamodel management with collection-based operators. In *Proc. MoDELS*, pages 236–245. IEEE Computer Society.

[Sammet, 1972] Sammet, J. E. (1972). Programming languages: History and future. *Commun. ACM*, 15(7):601–610.

[Sannier et al., 2013] Sannier, N., Acher, M., and Baudry, B. (2013). From comparison matrix to variability model: The wikipedia case study. In *Proc. ASE*, pages 580–585. IEEE.

[Sarencheh and Schiffauerova, 2017] Sarencheh, S. and Schiffauerova, A. (2017). Automatic algorithm for extracting an ontology for a specific domain name. In *Proc. KEOD*, pages 49–56. SciTePress.

[Sarimbekov et al., 2016] Sarimbekov, A., Stadler, L., Bulej, L., Sewe, A., Podzimek, A., Zheng, Y., and Binder, W. (2016). Workload characterization of JVM languages. *Software: Practice and Experience*, 46(8):1053–1089.

[Schaffert et al., 2006] Schaffert, S., Bischof, D., Bürger, T., Gruber, A., Hilzensauer, W., and Schaffert, S. (2006). Learning with semantic wikis. In *Proc. SemWiki2006@ESWC*, volume 206 of *CEUR Workshop Proceedings*. CEUR-WS.org.

[Schauss et al., 2017] Schauss, S., Lämmel, R., Härtel, J., Heinz, M., Klein, K., Härtel, L., and Berger, T. (2017). A chrestomathy of DSL implementations. In *Proc. SLE*, pages 103–114. ACM.

[Scherp et al., 2011] Scherp, A., Saathoff, C., Franz, T., and Staab, S. (2011). Designing core ontologies. *Applied Ontology*, 6(3):177–221.

[Schwarzer et al., 2016] Schwarzer, M., Schubotz, M., Meuschke, N., Breitinger, C., Markl, V., and Gipp, B. (2016). Evaluating link-based recommendations for wikipedia. In *Proc. JCDL*, pages 191–200. ACM.

[Seibel et al., 2010] Seibel, A., Neumann, S., and Giese, H. (2010). Dynamic hierarchical mega models: comprehensive traceability and its efficient maintenance. *Software and System Modeling*, 9(4):493–528.

[Seifer et al., 2019] Seifer, P., Härtel, J., Leinberger, M., Lämmel, R., and Staab, S. (2019). Empirical study on the usage of graph query languages in open source Java projects. In *Proc. SLE*, pages 152–166. ACM.

[Seybold et al., 2016] Seybold, D., Domaschka, J., Rossini, A., Hauser, C. B., Griesinger, F., and Tsitsipas, A. (2016). Experiences of models@run-time with EMF and CDO. In *Proc. SLE*.

[Sfar et al., 2016] Sfar, H., Chaïbi, A. H., Bouzeghoub, A., and Ghezala, H. B. (2016). Gold standard based evaluation of ontology learning techniques. In *Proc. SAC*, pages 339–346. ACM.

[Shatnawi et al., 2017] Shatnawi, A., Mili, H., El-Boussaidi, G., Boubaker, A., Guéhéneuc, Y., Moha, N., Privat, J., and Abdellatif, M. (2017). Analyzing program dependencies in Java EE applications. In *Proc. MSR*, pages 64–74. IEEE Computer Society.

[Shilov et al., 2012] Shilov, N. V., Akinin, A. A., Zubkov, A. V., and Idrisov, R. I. (2012). Development of the Computer Language Classification Knowledge Portal. In *Proc. PSI*, volume 7162 of *Lecture Notes in Computer Science*, pages 340–348. Springer.

[Simmonds et al., 2015] Simmonds, J., Perovich, D., Bastarrica, M. C., and Silvestre, L. (2015). A megamodel for Software Process Line modeling and evolution. In *Proc. MoDELS*, pages 406–415. IEEE Computer Society.

[Sindhgatta, 2008] Sindhgatta, R. (2008). Identifying domain expertise of developers from source code. In *Proc. SIGKDD*, pages 981–989. ACM.

[Smith and Welty, 2001] Smith, B. and Welty, C. A. (2001). FOIS introduction: Ontology - towards a new synthesis. In *FOIS*, pages iii–ix. ACM.

[Sobernig et al., 2011] Sobernig, S., Gaubatz, P., Strembeck, M., and Zdun, U. (2011). Comparing complexity of API designs: an exploratory experiment on dsl-based framework integration. In *Proc. GPCE*, pages 157–166. ACM.

[Solanki et al., 2016] Solanki, M., Božić, B., Freudenberg, M., Kontokostas, D., Dirschl, C., and Brennan, R. (2016). *Enabling Combined Software and Data Engineering at Web-Scale: The ALIGNED Suite of Ontologies*, volume 9982 of *Lecture Notes in Computer Science*, pages 195–203. Springer.

[Sottet et al., 2009] Sottet, J., Calvary, G., Favre, J., and Coutaz, J. (2009). Megamodeling and metamodel-driven engineering for plastic user interfaces: MEGA-UI. In *Human-Centered Software Engineering - Software Engineering Models, Patterns and Architectures for HCI*, Human-Computer Interaction Series, pages 173–200. Springer.

[Sottet et al., 2018] Sottet, J., Grandry, E., and Bjekovic, M. (2018). Managing regulatory system with megamodelling. In *Proc. CBI*, pages 1–10. IEEE Computer Society.

[Staab and Maedche, 2001] Staab, S. and Maedche, A. (2001). Knowledge portals: Ontologies at work. *AI Magazine*, 22(2):63–75.

[Staab et al., 2010] Staab, S., Walter, T., Gröner, G., and Parreiras, F. S. (2010). Model driven engineering with ontology technologies. In *Proc. Reasoning Web. Semantic Technologies for Software Engineering*, volume 6325 of *Lecture Notes in Computer Science*, pages 62–98. Springer.

[Steinberg et al., 2008] Steinberg, D., Budinsky, F., Merks, E., and Paternostro, M. (2008). *EMF: eclipse modeling framework*. Pearson Education.

[Stevens, 2017] Stevens, P. (2017). Bidirectional transformations in the large. In *Proc. MODELS*, pages 1–11. IEEE Computer Society.

[Stevens, 2018] Stevens, P. (2018). Towards sound, optimal, and flexible building from megamodels. In *Proc. MODELS*, pages 301–311. ACM.

[Stevens, Perdita, 2019] Stevens, Perdita (2019). Maintaining consistency in networks of models: bidirectional transformations in the large. *Software & Systems Modeling*, 19(1):39–65.

[Strom, 2000] Strom, S. (2000). Building a large-scale generic object model: applying the CYC upper ontology to object database development in java. In *Proc. OOPSLA*, pages 37–38. ACM.

[Strube and Ponzetto, 2006] Strube, M. and Ponzetto, S. P. (2006). Wikirelate! computing semantic relatedness using wikipedia. In *Proc. AAAI*, pages 1419–1424. AAAI Press.

[Stvilia et al., 2008] Stvilia, B., Twidale, M. B., Smith, L. C., and Gasser, L. (2008). Information quality work organization in wikipedia. *JASIST*, 59(6):983–1001.

[Subramanian et al., 2014] Subramanian, S., Inozemtseva, L., and Holmes, R. (2014). Live API documentation. In *Proc. ICSE*, pages 643–652. ACM.

[Sure et al., 2002] Sure, Y., Angele, J., and Staab, S. (2002). Ontoedit: Guiding ontology development by methodology and inferencing. In *Proc. DOA/CoopIS/ODBASE*, volume 2519 of *Lecture Notes in Computer Science*, pages 1205–1222. Springer.

[Sure et al., 2009] Sure, Y., Staab, S., and Studer, R. (2009). Ontology engineering methodology. In *Handbook on Ontologies*. Springer.

[Tanon et al., 2016] Tanon, T. P., Vrandecic, D., Schaffert, S., Steiner, T., and Pintscher, L. (2016). From freebase to wikidata: The great migration. In *Proc. WWW*, pages 1419–1428. ACM.

[Torres et al., 2012a] Torres, D., Molli, P., Skaf-Molli, H., and Díaz, A. (2012a). From DBpedia to Wikipedia: Filling the gap by discovering wikipedia conventions. In *Proc. Web Intelligence*, pages 535–539. IEEE Computer Society.

[Torres et al., 2012b] Torres, D., Molli, P., Skaf-Molli, H., and Díaz, A. (2012b). Improving Wikipedia with DBpedia. In *Proc. WWW*, pages 1107–1112. ACM.

[Toure et al., 2017] Toure, E. H. B., Fall, I., Bah, A., Camara, M. S., and Ba, M. (2017). Consistency preserving for evolving megamodels through axiomatic semantics. In *Proc. ISCV*, pages 1–8. IEEE.

[Tran and Debruyne, 2012] Tran, T. and Debruyne, C. (2012). Towards Using OWL Integrity Constraints in Ontology Engineering. In *Proc. OTM*, volume 7567 of *Lecture Notes in Computer Science*, pages 282–285. Springer.

[Treude and Robillard, 2017] Treude, C. and Robillard, M. P. (2017). Understanding stack overflow code fragments. In *Proc. ICSME*, pages 509–513. IEEE Computer Society.

[Troya et al., 2018] Troya, J., Segura, S., Parejo, J. A., and Cortés, A. R. (2018). Spectrum-based fault localization in model transformations. *ACM Trans. Softw. Eng. Methodol.*, 27(3):13:1–13:50.

[Tufano et al., 2017] Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Penta, M. D., Lucia, A. D., and Poshyvanyk, D. (2017). When and why your code starts to smell bad (and whether the smells go away). *IEEE Trans. Software Eng.*, 43(11):1063–1088.

[Tzekou et al., 2011] Tzekou, P., Stamou, S., Kirtsis, N., and Zotos, N. (2011). Quality assessment of wikipedia external links. In *Proc. WEBIST*, pages 248–254. SciTePress.

[Uddin et al., 2012] Uddin, G., Dagenais, B., and Robillard, M. P. (2012). Temporal analysis of API usage concepts. In *Proc. ICSE*, pages 804–814. IEEE Computer Society.

[Uddin and Robillard, 2015] Uddin, G. and Robillard, M. P. (2015). How API documentation fails. *IEEE Software*, 32(4):68–75.

[Urbani et al., 2009] Urbani, J., Kotoulas, S., Oren, E., and van Harmelen, F. (2009). Scalable distributed reasoning using mapreduce. In *Proc. ISWC*, volume 5823 of *Lecture Notes in Computer Science*, pages 634–649. Springer.

[Vadim Zaytsev, 2011] Vadim Zaytsev (2011). MediaWiki Grammar Recovery. *Computing Research Repository (CoRR)*.

[Vadim Zaytsev, 2017] Vadim Zaytsev (2017). Megamodelling with NGA Multimodels. In *Proc. CoCoS*, pages 1–6.

[van Roy, 2009] van Roy, P. (2009). Programming Paradigms for Dummies: What Every Programmer Should Know. volume 104, pages 616–621.

[Varanovich, 2018] Varanovich, A. (2018). *Software chrestomathy as a knowledge-driven research infrastructure for software engineering*. PhD thesis, Universität Koblenz-Landau.

[Vignaga et al., 2009] Vignaga, A., Jouault, F., Bastarrica, M. C., and Brunelière, H. (2009). Typing in model management. In *Proc. ICMT*, volume 5563 of *Lecture Notes in Computer Science*, pages 197–212. Springer.

[Vignaga et al., 2013] Vignaga, A., Jouault, F., Bastarrica, M. C., and Brunelière, H. (2013). Typing artifacts in megamodeling. *Software and System Modeling*, 12(1):105–119.

[Vogel and Giese, 2012] Vogel, T. and Giese, H. (2012). A language for feedback loops in self-adaptive systems: Executable runtime megamodels. In *Proc. SEAMS*, pages 129–138. IEEE Computer Society.

[Vogel and Giese, 2014] Vogel, T. and Giese, H. (2014). Model-driven engineering of self-adaptive software with EUREMA. *TAAS*, 8(4):18:1–18:33.

[Vogel et al., 2010] Vogel, T., Seibel, A., and Giese, H. (2010). The role of models and megamodels at runtime. In *Proc. MODELS*, volume 6627 of *Lecture Notes in Computer Science*, pages 224–238. Springer.

[Völkel et al., 2006] Völkel, M., Krötzsch, M., Vrandecic, D., Haller, H., and Studer, R. (2006). Semantic wikipedia. In *Proc. WWW*, pages 585–594. ACM.

[Vrandecic, 2012] Vrandecic, D. (2012). Wikidata: a new platform for collaborative data collection. In *Proc. WWW*, pages 1063–1064. ACM.

[Walter et al., 2009] Walter, T., Parreiras, F. S., and Staab, S. (2009). *OntoDSL*: An ontology-based framework for domain-specific languages. In *Proc. MODELS*, volume 5795 of *Lecture Notes in Computer Science*, pages 408–422. Springer.

[Wang et al., 2010] Wang, H., Jiang, X., Chia, L., and Tan, A. (2010). Wikipedia2Onto. Building Concept Ontology Automatically, Experimenting with Web Image Retrieval. *Informatica (Slovenia)*, 34(3):297–306.

[Wang et al., 2012] Wang, S., Lo, D., and Jiang, L. (2012). Inferring semantically related software terms and their taxonomy by leveraging collaborative tagging. In *Proc. ICSM*, pages 604–607. IEEE Computer Society.

[Wang et al., 2018] Wang, S., Lo, D., Vasilescu, B., and Serebrenik, A. (2018). Entagrec ++: An enhanced tag recommendation system for software information sites. *Empirical Software Engineering*, 23(2):800–832.

[Wohlin, 2014] Wohlin, C. (2014). Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proc. EASE*, pages 38:1–38:10. ACM.

[Wu and Weld, 2008] Wu, F. and Weld, D. S. (2008). Automatically refining the wikipedia infobox ontology. In *Proc. WWW*, pages 635–644. ACM.

[Wu and Weld, 2010] Wu, F. and Weld, D. S. (2010). Open information extraction using Wikipedia. In *Proc. ACL*, pages 118–127. The Association for Computer Linguistics.

[Yu et al., 2007] Yu, J., Thom, J. A., and Tam, A. M. (2007). Ontology evaluation using wikipedia categories for browsing. In *Proc. CIKM*, pages 223–232. ACM.

[Zarrad et al., 2013] Zarrad, R., Doggaz, N., and Zagrouba, E. (2013). Title-based approach to relation discovery from wikipedia. In *Proc. KEOD*, pages 70–80. SciTePress.

[Zaveri et al., 2013] Zaveri, A., Kontokostas, D., Sherif, M. A., Bühmann, L., Morsey, M., Auer, S., and Lehmann, J. (2013). User-driven quality evaluation of dbpedia. In *Proc. ISEM*, pages 97–104. ACM.

[Zaytsev, 2011] Zaytsev, V. (2011). Mediawiki grammar recovery. *CoRR*.

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

[Zaytsev, 2014] Zaytsev, V. (2014). Understanding Metalanguage Integration by Renarrating a Technical Space Megamodel. In *Proc. GEMOC@Models*, volume 1236 of *CEUR Workshop Proceedings*, pages 69–77.

[Zaytsev and Bagge, 2014] Zaytsev, V. and Bagge, A. H. (2014). Parsing in a Broad Sense. In *Proc. MODELS*, volume 8767 of *Lecture Notes in Computer Science*, pages 50–67.

[Zerouali and Mens, 2017] Zerouali, A. and Mens, T. (2017). Analyzing the evolution of testing library usage in open source Java projects. In *Proc. SANER*, pages 417–421. IEEE.

[Zhang et al., 2010] Zhang, N., Huang, G., Zhang, Y., Jiang, N., and Mei, H. (2010). Towards automated synthesis of executable eclipse tutorials. In *Proc. SEKE*, pages 591–598. Knowledge Systems Institute Graduate School.

[Zhao et al., 2014] Zhao, D., Liu, P., Qin, L., and Li, Y. (2014). A novel terms semantic query model based on wikipedia. In *Proc. WISA*, pages 258–261. IEEE Computer Society.