A Rule-based Approach for the Automatic Extraction of Mega Models

Masterarbeit

zur Erlangung des Grades eines Master of Science
im Studiengang MSc. Informatik

vorgelegt von

Frederik Rüther

Erstgutachter:	Prof. Dr. Ralf Lämmel Institut für Informatik
Zweitgutachter:	Msc. Johannes Härtel Institut für Informatik

Koblenz, im Februar 2018

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

.....

(Ort, Datum) (Frederik Rüther)

Zusammenfassung

Moderne Softwaresysteme bestehen aus verschiedenen Programmiersprachen, Softwaretechnologien und Artefakten. Dadurch wird es für Entwickler komplexer, den Quelltext sowie die enthaltenen Abhängigkeiten zu verstehen. Entsprechend muss ein größerer Aufwand in die Erstellung von Dokumentation gesteckt werden. Eine Möglichkeit zur Dokumentation einer Software mit dem Fokus auf die benutzten Technologien stellen linguistische Architekturen dar. Diese können z. B. durch die *MegaL* Ontologie beschrieben werden. Da die Erstellung einer solchen linguistischen Architektur für ein beliebiges Softwareprojekt kompliziert ist, beschreibt diese Arbeit einen Ansatz zur automatischen Erstellung einer solchen linguistischen Architektur. Dafür wird das Open Source Framework Apache Jena verwendet, welches Semantic Web Technologien wie RDF und OWL benutzt. Mit diesem können spezifische Regeln definiert werden, welche aus existierenden RDF-Triplen neue ableiten. Dieser Ansatz wird schließlich in einer Case Study an zehn verschiedenen Open Source Projekten getestet. Dabei soll eine linguistische Architektur in *MegaL* extrahiert werden, welche die Nutzung von Hibernate beschreibt. Mit der Hilfe von spezifischen Metriken wird das Ergebnis dann mit einem internen und externen Ansatz evaluiert.

Abstract

Modern software projects are composed of several software languages, software technologies and different kind of artifacts. Therefore, the understanding of the software project at hand, including the semantic links between the different parts, becomes a difficult challenge for a developer. One approach to attack this issue is to document the software project with the help of a linguistic architecture. This kind of architecture can be described with the help of the *MegaL* ontology. A remaining challenge is the creation of it since it requires different kind of skills. Therefore, this paper proposes an approach for the automatic extraction of a linguistic architecture. The open source framework Apache Jena, which is focusing on semantic web technologies like RDF and OWL, is used to define custom rules that are capable to infer new knowledge based on the defined or already extracted *RDF triples*. The complete approach is tested in a case study on ten different open source projects. The aim of the case study is to extract a linguistic architecture that is describing the use of Hibernate in the selected projects. In the end, the result is evaluated with the help of different metrics. The evaluation is performed with the help of an internal and external approach.

Acknowledgement

I would like to thank my supervisors Prof. Dr. Ralf Lämmel and Johannes Härtel for their support during the progress of creating this thesis. Furthermore, I want to thank the complete *Softlang* team for their advice, tips and learning opportunities during my studies in the MSc. Computer Science program of the University of Koblenz.

Contents

1 Introduction **1**

2 Related Work **3**

3 Background **5**
3.1 Ontologies 5
3.2 Linguistic Architectures 5
3.3 Entities in MegaL 5
3.4 Relations in MegaL 7

4 A Rule-based Approach for Mega Model Extraction **9**
4.1 Overview 9
4.2 Preparations 10
4.3 Rule-based Inference of Mega Models 11

5 Case Study **15**
5.1 Methodology for Technology and Project Selection 15
5.2 Target Mega Model for Hibernate 18
5.3 Iterative Rule Development 24
5.4 Evaluation 36
5.5 Discussion 45

6 Concluding Remarks **47**
6.1 Summary 47
6.2 Threat to Validity 47
6.3 Future Work 48

7 Appendix **50**
7.1 SQL commands 50
7.2 Further MegaL Definitions 52
7.3 API description of Hibernate 54

References **55**

List of Tables

4 Basic entities of MegaL 6
5 MegaL relations used by our approach 8
6 Software projects that fulfill the criteria 18
7 Shows the methods used to identify a language 25
8 Services that can be used by the built-ins 35
9 The developed metrics for the evaluation 37
10 Summary of the result for the internal validation 40
11 Comparison of the results between the tool and naive approach 42

12 The value of the metric and the result of the search of the DAO files with the naive (N) approach . . . 43
13 Shows how many files, not covered by the naive approach, are correctly and wrongly classified by the tool . . . 43
14 Table statement, correspondences and mapping definitions of the projects 46
16 Entities depending on the ecosystem used by software engineering 52

List of Figures

1 Links between O/R-Mapping related artifacts . . . 1
2 Logical process for the extraction of a linguistic architecture . . . 9
3 Most used APIs of maven projects on GitHub according to [14] . . . 16
4 The extraction process used by our approach . . . 24
5 The relation between rules, built-ins and services . . . 35

Listings

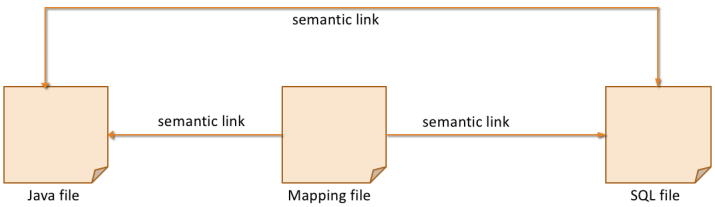
1 Encoding of a MegaL Role and Language entity into RDF . . . 10
2 Definition of artifacts found in the project folders . . . 10
3 Packages encoded into MegaL vocabulary . . . 11
4 Example of a rule with a body built-in . . . 12
5 Content of an exemplary XML file . . . 12
6 Example of another rule with a body built-in . . . 13
7 Example of a rule with a head built-in . . . 13
8 Triples added by the built-in . . . 13
9 Backward rule for the finding of no values . . . 14
10 An exemplary patient class . . . 18
11 Definition of mappable objects in MegaL . . . 19
12 An exemplary create statement for a patient table . . . 20
13 Definition of the relational model in MegaL . . . 20
14 Relation of interest . . . 20
15 Definition of the specific mapping function . . . 20
16 An example of a Hibernate mapping description . . . 21
17 Definition of the object specific mapping function . . . 21
18 The annotated patient class . . . 22
19 Definition of the specific mapping function with Java annotations . . . 22
20 Example of a service which is executing a patient mapping function . . . 22
21 The service data encoded into MegaL . . . 23
22 The different roles found by using the Hibernate technology . . . 23
23 Identification of the language of an artifact . . . 25
24 Fragment extraction and parsing for a SQL create statements . . . 26
25 Result of the rule shown in Listing 24 . . . 26

26 Checking if a XML file conforms to a XSD file 26
27 Extraction of the Java class URI 27
28 Hibernate role identification with the help of a built-in 27
29 Hibernate role identification with the help of a built-in 28
30 Mapping definitions with the help of annotations 29
31 The result of the execution of the rule shown in Listing 30 29
32 Definition of the OLang for each annotated class 30
33 Definition of the mapping function 30
34 The linking of the function to its input and output 31
35 The linking of the function to its input and output 32
36 Detection of inheritance between different persistable objects 34
37 This fragment is an example for a false positive error 45
38 This fragment is an example for a false negative case 45
39 Getting final hibernate versions 50
40 Getting projects with fitting versions and classes 50
41 SQL for linking projects with API usage 50
42 Multiple persons should work on a Hibernate project 51

1 Introduction

Motivation:
Modern software systems are composed of different technologies, languages and frameworks. This leads to technology specific artifacts in software projects like for example a Hibernate O/R-Mapping description file. A developer working on a system can quickly loose the overview about the dependencies between different artifacts established by the used technologies. Especially, in the context of software maintenance the process of understanding a software system becomes a main challenge. The importance is underlined by the estimation that up to 50 % of the maintenance effort is spend on understanding source code [10].
One approach to address this problem is to create a linguistic architecture that illustrates the used technologies, languages and artifact of a software system [5, 9]. This way a software system can be comprehended regarding the usage of a specific technology and finally show the semantic links between different files. Let's for example consider a software system that is using a Hibernate mapping description to map an object to a relational table. The latter one is defined in a SQL file. In this usage scenario, a semantic connection, as shown in Listing 1, between the *Java* and *SQL* file exists since both describe the same data but with a different language. In addition, the mapping file is indirectly connected to the files since it is referencing their class and table name. These kind of relations and others describing the language and technology usage can be illustrated with the help of a *linguistic architecture* and finally help the developer to understand the software system.

Figure 1 Links between O/R-Mapping related artifacts



The manual creation process of a *linguistic architecture* is time consuming and requires domain knowledge of the project. In addition, a developer would need experience with linguistic architectures and their ecosystem [8]. Still we believe that models of a linguistic architecture can serve as a documentation of software technologies and software projects. In other contexts, ontologies were already automatically extracted for the documentation and summary of software project [17, 10]. In fact, that field is known as ontology-enabled development (OED) and it aims to support developers while coding [7]. A linguistic architecture can be described by an ontology. The remaining challenge is the quick and correct creation of it, though.
Research questions:
Therefore, this paper tries to automatically extract the *linguistic architecture* of a software

system with the focus on documenting and describing the use of a specific technology. The key concept here is that the model should be automatically extracted. For the extraction of the linguistic architecture, a rule based approach is developed, that is using technologies and techniques from the semantic web context to define custom rules that infer new knowledge based on existing information. The linguistic architecture is formalized in the mega modelling language *MegaL*. The technology we are focusing on is Hibernate. In the end, this approach tries to answer the following two related research questions:

1. Can a linguistic architecture automatically be extracted?
2. Is it possible to find and investigate valuable characteristics of the software systems with the extracted model?

A case study on open source projects will be performed to answer these questions. It is focusing on the extraction of Hibernate related information on several open source projects.

Contributions:

This thesis has several contributions to the research community which are listed below:

- Creation of a linguistic architecture with the language *MegaL* for the description of the O/R-mapping process with the Hibernate framework.
- Development of an approach, based on *RDF* triples and rule based reasoning, for the automatic extraction of a linguistic architecture.
- A code base that supports the approach by providing functionality for the extraction of information from source files.
- An evaluation of the approach with the help of a case study on ten Java based open source projects that are using the Hibernate framework.

Structure of the thesis:

The roadmap of the paper is as following: First related work is discussed. Chapter 3 gives an introduction about required knowledge which is mainly focused on the term ontology and the mega modelling language *MegaL*. Afterwards, the rule based approach is introduced with the help of a placeholder technology. In Chapter 4 a case study with Hibernate as technology is performed. The case study does include a methodology for technology and project selection and in the end the result is evaluated and summarized. The thesis closes with concluding remarks about threats and future work.

2 Related Work

Ontologies in the context of software engineering:
An ontology is used in this paper for the representation of the linguistic architecture. There exist different scenarios in the research field of software engineering and software maintenance were ontologies are used. *Happel* and *Seedorf* [7] present some examples of ontology applications throughout the software engineering life cycle. As a result, a classification is discussed based on the roles the ontology is playing. Inspired by their classification schema our approach could serve as an example of *ontology-enabled development* since the extracted *linguistic architecture* should support developers with their tasks by providing descriptions about the technology usage in the system.
Meng et al. [10] are using an ontology and description logic to support the comprehension of a software system. Their approach aims to query a previous populated ontology and to apply formal reasoning to it. *Anquetil* et al. [2] see the challenges of software maintenance as a knowledge management issue or rather the lack of knowledge as a prominent problem. Therefore, they developed an ontology for software maintenance that is populated by the maintainers. Both papers are related to our approach since they are using an ontology to formalize knowledge about a software engineering related domain and finally enable reasoning based on the encoded information.
Our approach encodes as well information into an ontology to support developers and enables reasoning on the formalized information. In contrast, our approach is concerned with a different kind of ontology and tries to automatically populate it. The mentioned papers were manually populating an ontology.

Linguistic architectures:
The approach developed in this thesis is defining and extracting a linguistic architecture related to the usage of the Hibernate framework. A linguistic architecture is concerned with the usage scenarios of software technologies in terms of the involved software languages, related technologies and the linguistic relation they are having. *Favre* et al. [5] are demonstrating the creation of such a linguistic architecture for Object/XML mapping with the help of mega modelling. Based on this work *Lämmel* and *Varanovich* [9] are extending the mega modelling approach to provide a general facility to apply mega models to actual systems and to validate the claims that are made by the mega models. Finally, *Härtel* et al. [8] are trying to make *MegaL*, a proposed language for creating a linguistic architecture, more useful for developers by adding further features to it.
Inspired by these papers, we developed an approach for the automatic extraction of a linguistic architecture from a software system. Furthermore, the mega modelling language *MegaL* described in these papers is used for the formalization of the linguistic architecture. However, our focus is different since it lays on the automatic extraction of a linguistic architecture and does neither improve its concepts nor its ecosystem.

Semantic web technologies and software engineering:
The rule based approach we use for the population of the ontology is relying on semantic web technologies. Other papers in the field of software engineering are exploit-

ing these technologies in a similar way. For instance were semantic web technologies mapped by *Witte* et al. [16] to the field of software maintenance. Their approach created an united representation, which enables to explore, query and reason about a multitude of software artifacts. In the end, this should ease the work of software maintainers that have to deal with documentations, source code and their semantic connections. A key component was the automatic population of the ontology. This approach was extended by *Zhang* et al. [17] to improve the links between documentation and source code files by using text mining techniques. Finally, *Al-Qahtani* et al. [1] mapped the approach to the domain of software security. There they tried to automatically extract traceability links between APIs and software vulnerabilities.
These papers are related to our approach since they are as well using semantic web technologies, like RDF, to represent an ontology. On top of this, they are as well automatically populating an ontology based on the artifacts of a software system. However, there are two differences. On the one hand, the domain differs since we are applying the semantic web technologies to extract an ontology that focuses on technology usage and the semantic relations caused by it. On the other hand, they are not using rule based reasoning to populate the ontology.

API and technology focused analysis of software:
A linguistic architecture is one way to analyze and summarize the usage of technologies and languages in a software project. Several other techniques and research papers exist that are concerned with technologies and API usage in software projects. *Nagy* et al. [11] determined where a SQL statement was executed in a software project that is using the Hibernate framework. This paper is related since it is as well focusing on the technology Hibernate and on top of this tries to trace where a SQL query was created in the source code. This is a similar task to finding the usage of O/R-mapping functions that we want to perform.
Ratiu et al. are extracting a domain model in the form of an ontology by analyzing the relations inside a domain specific API. In contrast, our approach is as well extracting an ontology to analyze the relations in the software system but does not focus on domain specific APIs.
The evaluation of the approach described in this thesis relies on technology specific metrics that summarize the usage of the technology. This is related to the approach of *Roover* et al. [13] that is analyzing with the help of metrics the usage of APIs in software projects, e.g., the amount of found API elements.
A linguistic architecture of a software system is as well extracted by *Favre* et al. [4]. In fact, their approach is extracting from source files of a software chrestomathy relevant information and finally assigns those as meta data to the artifacts. In contrast, this thesis focuses on an approach that works on arbitrary open source projects and develops a rule based approach to extract a linguistic model for a Hibernate use case.

3 Background

Before the approach can be explained in detail an explanation and definition of the used terms and vocabulary has to be provided to clarify what is meant and how they relate to each other. Therefore, a definition for the term *ontology* and *linguistic architecture* is given in chapter 3.1 and 3.2 . Finally, the vocabulary used by the language *MegaL* is explained in detail in chapters 3.3 and 3.4.

3.1 Ontologies

Ontologies can be used for the definition of a linguistic architecture. An ontology is defined by *Uschold* and *Gruninger* [15] as a term used *"to refer to the shared understanding of some domain of interest"*. In the context of this research, the domain is the field of software engineering. They continue that an ontology is often *"conceived as a set of concepts, their definitions and their inter-relationships"* [15]. This means that the different concepts can exist in an ontology like for example a *Person* and a *Car*. Both concepts can now be connected by defined relations. A relationship that could connect these two concepts could be named *owns*, which would indicate that a specific instance of a person is owning an instance of the concept car, e.g., a Person called *Peter* could be connected to the car *Audi-A6* by this relationship: *Peter owns Audi-A6*.

3.2 Linguistic Architectures

A linguistic architecture is concerned with the usage of software languages, software technologies and artifacts in a software system and tries to lay out the relations between those different components [5]. The models of linguistic architectures can be referred to as mega models. One language to describe a mega model is called *MegaL* and it builds an ontology that defines a set of entities/concepts and relation types [5].
There are several concerns a linguistic architecture wants to address. On the one hand, it can be used as a documentation for developers to understand how to use a technology in a different scenario. On the other hand, it can be used to understand how technologies are used in a software project.

3.3 Entities in MegaL

The language *MegaL* is composed of several entities. An entity in *MegaL* can be seen as a *concept* of an ontology as it was introduced in Chapter 3.1. Each of the explained elements in this chapter are sub elements of an *Entity* in *MegaL*. The official *Megalib*[1] repository and the published papers [9, 5] serve as a reference for the details explained in this chapter.
Artifacts in the *MegaL* ontology are elements with a physical manifestation. An example of an artifact could be a file or a folder. Artifacts can further be splitted into different *Fragments*. A *Fragment* refers to a continues range of an artifact, e.g., a method definition

[1] https://github.com/softlang/megalib

could be an example of a fragment of a Java file. Some *Artifact* have a special purpose or task in a software system. This is described by the entity *Role*.
Another important entity is *Language*, which is conceptually a special type of a set. Java is an example for an instance of such an entity but also domain specific or embedded languages can exist.
A software project is in the *MegaL* context described as a *System*. Other notable examples of *Systems* are technologies, libraries, APIs, frameworks and applications.
One aim of mega modelling is to describe technologies and their implemented features. A way to do this is to define a *Function* that describes the meaning of and the operations performed by the technology. This is enabled by *MegaL* with the corresponding entity name. A *Function* can take several *Artifacts* of a specific language as input and returns an *Artifact* of a defined language. Table 4 is showing the discussed entities of *MegaL*. The language itself defines more. However, just the ones necessary for this approach were introduced.

Name	**Description**	**Origin**	**Hyperlink**
Artifact	This is any kind of digital entity created during the software engineering process.	MegaL [9]	Wikipedia
Language	In simple terms it is may described as a set	MegaL [9]	Wikipedia
Fragment	Is a continues region of a given artifact, e.g., line number 10 to 20.	MegaL [9]	Wikipedia
System	Is a reusable third party software. Technologies, APIs, libraries and even normal applications are examples of a System	Megalib repository	Wikipedia
Function	The meaning of a program or system, e.g, a certain feature or operation that a system is able to perform. Usually it is composed of input elements and returns a result.	MegaL [9]	Wikipedia
Role	The purpose, task or classification of a software artifact. Usually it is describing the special purpose the artifact is playing in the technology.	Megalib repository	Wikipedia

Table 4 Basic entities of MegaL

3.4 Relations in MegaL

The different entities that are defined in the *MegaL* ontology can be connected with relations. The most important relations are explained inside Table 5.
The most generic relation is named *partOf* and it describes the composition of two elements, e.g., a *Fragment* is *partOf* an *Artifact*.
Language specific: If an *Artifact* is written in a specific *Language* both instances of the entities are connected by the relation *elementOf*. An example would be a Java file which is an *elementOf* the language *Java*. A *Language* itself has to be defined, e.g., by a grammar specification or source code. This is described by the relation called *defines*, which connects an *Artifact* to the *Language* it specifies. If an *Artifact* is not just the element of a *Language* but it is also following a certain schema, it *conformsTo* the artifact describing the schema. A *XML* file would usually *conformTo XSD*.
Content specific: Other relations are linking entities based on the content of artifacts. This can happen if either the content is referencing another artifact e.g., by an URI or when the content is specifying a certain construct of the language like a class or a table. The most interesting relation for this thesis is *correspondsTo*. This relationship indicates that both entities describe the same data but with a different language or abstraction level. Take for example an employee of a company with all its properties. This data could be described by a XML or JSON file. If both files exists they would *correspondTo* each other since they are describing the same data, an employee, but with a different encoding.
Technology specific: If technologies are used in a project they usually lead to specific relations. A technology quite often manifests *Artifacts* that play a certain *Role* in a scenario. An example would be a file having the *Role* of a *Driver*. The relation *hasRole* would connect the file to its specific *Role*. A technology is usually implementing a function. In the context of *MegaL*, such a function is just accepting input of a certain kind. Therefore, the *Languages* expected as input and output are defined. A concrete scenario can be described by specifying the *Artifacts* involved in the function execution as argument and the result. However, they have to be an *elementOf* the required *Language*.

Name	Type definition	Description	Origin
partOf	Artifact # System Function # System Fragment # Artifact Function # Function	Basically indicates that an element is composed of other elements.	MegaL [5]
elementOf	Artifact # Language	An artifact that is written in a certain language is an element of this particular language.	MegaL [5]
defines	Artifact # Language Artifact # Function	An artifact can define a language	MegaL [5]

conformsTo	Artifact # Artifact	An artifact may follows a certain schema. If this schema is defined in another artifact the artifacts conform to each other.	MegaL [5]
corresponsTo	Artifact # Artifact	This link indicates that both objects describe the same data or facts but with different a encoding	MegaL [5]
hasRole	Artifact # Role	An artifact may has a certain purpose in this context	MegaL [5]

Table 5 MegaL relations used by our approach

4 A Rule-based Approach for Mega Model Extraction

This chapter aims to explain the rule based approach that is used for the extraction of a linguistic architecture. The approach is explained with the help of the placeholder technology XML and the linguistic architecture it manifests. In a case study the approach will later be executed and evaluated with a specific technology.

4.1 Overview

The linguistic architecture is extracted with the help of developed rules. These rules derive new knowledge based on the already existing information.
The execution and reasoning process is supported by the open source framework Apache Jena [2], which is designed for working with linked data in the RDF or OWL format. One part of Jena is a *generic rule reasoner* [3], that enables to build user defined rules to infer new *RDF triples* based on the already existing *RDF triples*. The existing triples are loaded to the reasoner as initial models. Since Jena is working on *RDF triples* the *MegaL* ontology has to be encoded into RDF. This is possible since *MegaL* uses mostly the *Entity*, *Relation*, *Entity* format, which can easily be encoded into *RDF triples* that support the subject–predicate–object layout. Furthermore, the *Entities* are encoded into an URI that starts with *http://softlang.com/*.
The overall approach for the extraction of the mega model is composed of three separate phases. Each phase is executed by a tool, which is developed by us and is manifesting an incremental architecture. The first step is to load an initial model for the reasoning process composed of *RDF triples* that are representing the software project and technologies at hand. It contains information like the *Artifacts* in a software project and *Role* definitions of the technologies. Chapter 4.2 gives an in depth explanation regarding this step of the approach. Afterwards, these triples will be used to extract the linguistic architecture with the help of custom rules. In the end, the complete linguistic architecture is summarized with the help of custom metrics and the mega model is displayed. Chapter 4.3 explains the Jena rule inference which is used for both phases. Figure 2 illustrations the logical steps of the approach.

Figure 2 Logical process for the extraction of a linguistic architecture

[2] https://jena.apache.org/
[3] https://jena.apache.org/documentation/inference/rules

4.2 Preparations

A model, basically a set of triples on which the reasoning should begin, has to be created. This model is composed of *RDF triples*, which can be distributed over different files. These triples are either added manually or automatically to the corresponding files. For each technology, like Hibernate, Spring or Jaxb, a specific folder exists where related files and basic triples are stored. Another directory exists for the software project to be analyzed. The files defining *RDF triples* in these folders will be loaded to build the initial model.
The *Roles* that appear in a technology are added manually in the form of *RDF triples* by the developers. Therefore, the name of the role itself as well as the *MegaL* relation and entity have to be encoded into an URI to form a valid triple. In the context of XML an example of a *Role* would be a *Schema*. A *XSD* file would have such a role since it defines the schema of specific *XML* files. Both information are encoded into an URI and as *RDF triples*. The type of an entity will be defined by the RDF related verb *type*. Another example of a triple that has to be added manually would be the definition of the language itself. Listing 1 illustrates the added *RDF triples*. Further, entities can be added by following the same steps.
Packages, that are part of the technology, are added manually to a *JSON* file that summarizes the packages of a technology.

Listing 1 Encoding of a MegaL Role and Language entity into RDF

```
1 @prefix sl: <http://softlang.com/> .
2 @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema> .
3
4 sl:Role/Schema rdfs:type sl:Role .
5 sl:Language/XML rdfs:type sl:Language .
```

The *Artifacts* of a software system or technology are added automatically as triples to the corresponding file. This is possible since the software system to be analyzed is located at the local file system. Therefore, the program iterates over the folders and extracts *RDF triples* defining the contained files of the project. For example would a XML file of the project with the name *employee.xml* be encoded into *RDF triples* as shown in Listing 2.
Technology specific artifacts, as for example the XSD or DTD schema files, are located in a previous defined technology folder. Accordingly the *RDF triples* for those files are created and stored automatically. The JSON file with the packages will be translated to *RDF triples* that define a *Package*, which is a sub-type of an *Artifact*, and be added to a model file. In the ecosystem of *XML* the technology *Jaxb* can be used for the mapping of XML data to Java objects. One of its *Packages* serves as an example. The result is illustrated in Listing 3.
The process closes by loading all the relevant files, containing the manual and automatic created *RDF triples*, to the Jena reasoner. These loaded triples are referred to as initial model.

Listing 2 Definition of artifacts found in the project folders

```
@prefix sl: <http://softlang.com/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema> .

sl:Project/Path/To/employee.xml rdfs:type sl:Artifact.
sl:Project/Path/To/employee.xml sl:partOf sl:Application/Project.
sl:Application/Project rdfs:type sl:System.
```

Listing 3 Packages encoded into MegaL vocabulary

```
@prefix sl: <http://softlang.com/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema> .

sl:Package/javax.xml.bind rdfs:type sl:Package.
sl:Package/javax.xml.bind sl:partOf sl:Technology/Jaxb.
sl:Technology/Jaxb rdfs:type sl:System.
sl:Package sl:subtype sl:Artifact.
```

4.3 Rule-based Inference of Mega Models

Once the *RDF triples*, as explained in chapter 4.2, are loaded into the reasoner the inference of further triples can start. This is performed by custom defined rules, that are depending on the (software) technology usage that should be described by the mega model. A mega model that focuses on the usage of the technology Hibernate manifests different rules than when the usage of the Spring framework should be documented. This chapter uses the *XML* ecosystem as a running example to explain the general approach and concepts used for the definition of the rules. A concrete definition of rules for a certain technology are described in the case study.

Rule Engine: The general purpose rule engine of Jena can be used to define rules that reason about existing *RDF triples*. These rules will have a body and a head that is composed of a list of terms. The body is defining the premises for the execution of a rule whereas the head defines the conclusions of a rule. A term can either be a triple or a built-in, that executes Java source code.

Forward engine: Jena has a forward and a backward engine for the execution of the rules. The forward engine is used for the inference of new triples and is based on the RETE algorithm [6]. Therefore, the rules are executed incrementally based on the inferred triples. A forward rule separates the body and head by a right arrow $\rightarrow$.

Sticking to our running example of XML, the first step would be to identify all files that are actually encoded into XML. The normal *triple based* reasoning is itself not expressive enough to extract such a complex relation. Therefore, Jena provides the possibility to develop built-ins in Java that can be used in the header or body of a forward-rule. This enables Jena to express difficult conditions. The built-ins are separated based on their position in the rule into head and body built-ins. A body built-in would be suitable to solve the described issue.

Body built-ins: These built-ins are part of the body of a rule. They are either returning

true or false and help to decide whether the list of triples in the head of the rule should be appended to the resulting set of triples.
For the described scenario a built-in, called *ParseXML*, could be coded, that takes a parameter as input and returns true if the referenced file could be parsed as XML. The parameter is the path of an *Artifact* encoded in an URI as shown in Listing 2. Listing 4 shows the complete rule for such a scenario. If the built-in returns true the triple in the conclusion is added to the set of triples in the model. This may enables the firing of another rule that was was requesting in the premise that a certain *Artifact* is an element of *XML*.

Listing 4 Example of a rule with a body built-in

```
1  (?a rdfs:type sl:Artifact)
2  ParseXML(?a)
3  ->
4  (?a sl:elementOf sl:Language/XML)
```

The next thing that is might interesting in a linguistic architecture is to define the *Role* of a certain *Artifact*. An approach for this could be to extract the name of a root element of a XML file and append the word *Description* to it. The result would be a meaningful name for a *Role*. Listing 5 shows the layout of a XML file that is describing an employee. Its root element is called *employee* as well. This name should be extracted by the built-in and be processed in a way that the output is *http://softlang.com/Role/EmployeeDescription*.
For this scenario it is not enough to just return true or false. It would in addition be good to bind the result to a variable so that it can be used in the rule afterwards. This should of cause just happen for XML files that are having a root element. Otherwise nothing should happen. This scenario can be realized with a body build-in since it allows to bind for example an URI to a variable that is given as an argument. Usually, this means that a previous *empty* variable does afterwards contain a certain URI.
A possible built-in with the name *CreateRoleName* would take two arguments. One would specify the path of a certain XML file and into the second argument the result is written. In case the XML file is empty, the premises are not true and the conclusion will not be added to the set of triples. Otherwise a new URI containing the *Role* is added. This rule is executed after the one defined in Listing 4 since the required triple is added there.

Listing 5 Content of an exemplary XML file

```
1 <employee snumber = 12>
2     <firstname> Hans </firstname>
3     <lastname> Mueller </lastname>
4     ...
5 </employee>
```

Listing 6 Example of another rule with a body built-in

```
(?a sl:elementOf sl:Language/XML)
CreateRoleName(?a, ?role)
->
(?a sl:hasRole ?role)
(?role rdfs:type sl:Role)
```

Head built-ins: A lot scenarios exist where more than one value has to be obtained and added as a result. Imagine the XML file should be split into different fragments where each fragment represents a XML-Element contained in the root. Looking at Listing 7 this would be the element *firstname* and *lastname*. This would may lead to multiple fragments that have to be added in the conclusion. This cannot be archived with a body built-in. Therefore, Jena provides another type of built-ins, that are executed in the head of a rule. Those will neither return true nor false but rather add calculated triples to the triple set. If the body found the requested triples, the head built-in will be executed, eventually with parameters bound by the body of the rule. For the realization of the described scenario the built-in *ExtractFragments* can be used that will extract all the elements of the XML file and adds them in the form of triples to the set. This way a XML file can easily be split into fragments. Listing 7 shows the complete rule and Listing 8 shows the expected result for the employee XML file.

Listing 7 Example of a rule with a head built-in

```
(?a sl:elementOf sl:Language/XML)
->
ExtractFragments(?a)
```

Listing 8 Triples added by the built-in

```
(sl:Project/Path/To/employee.xml:28#57 rdfs:type sl:Fragment)
(sl:Project/Path/To/employee.xml:62#92 rdfs:type sl:Fragment)
(sl:Project/Path/To/employee.xml:28#57 sl:partOf sl:Project/Path/To/employee.xml)
(sl:Project/Path/To/employee.xml:62#92 sl:partOf sl:Project/Path/To/employee.xml)
```

4.3.1 Evaluation Based on Metric Extraction

The last phase of the process tries to collect different metrics for the evaluation. The rule engine or *Jena* API can be used for the extraction of metrics. Those metrics depend on the chosen technology and linguistic architecture that should be extracted. They should answer questions about the analyzed software system and are developed with the specialties of a technology in mind. Eventually, it should be used to evaluate the quality of the developed rules for the extraction of the linguistic architecture. In the case of *XML* it would be of interest to count how many files do not conform to a certain schema. This metric will be extracted with the help of the backward engine of *Jena*.

Backward engine: The *backward engine* uses logic programming similar to the language Prolog. Its goal is to enable the querying of data. It is executed after the forward engine

finished its execution. This is especially useful if questions regarding the complete set of triples should be answered. In the XML example it could be of interest to count how many files exist, that have no *conformsTo* relation. In fact, a triple should be added to the set which identifies such files by tagging it with an object of the type *NoConformation*. This enables the counting of the elements with the help of the Jena API.
The *Jena* framework already provides a built-in called *noValue* which checks if a triple exists in the data set. Since this is executed after the forward engine all the existing triples of the forward engine should exist. As a result, it is safe to say that all possible triples were already inferred by the forward engine. Listing 9 shows the complete rule. A rule that is executed in the backward engine is identified by a left arrow ←. However, only body and not head built-ins can be used in this kind of rule.
In the end, these triples can be counted by the method *listSubjectsWithProperty* of the *Jena API* and populate a metric called *NotConformingFiles*.

Listing 9 Backward rule for the finding of no values

```
1  (?a sl:elementOf sl:Temp/NoConformation)
2  <-
3  (?x sl:elementOf sl:Language/XML)
4  noValue(?x, sl:conformsTo)
```

5 Case Study

5.1 Methodology for Technology and Project Selection

The approach we selected should be executed and evaluated in a case study which requires two different inputs. On the one hand, it has to be decided what technology should be focused on for the extraction of the linguistic architecture. On the other hand, sample projects using this technology have to be selected for the evaluation of the approach. This chapter describes how and with what criteria those elements have been selected. Its goal is to enable the reproducebility of the case study.

5.1.1 Technology Selection

The technology that should be selected for the case study has to fulfill the following two criteria:

1. The technology should be frequently used by projects.
2. Several languages should be used during the usage of the technology.

The first criteria ensures that the technology is popular and the extracted mega model is useful for a large group of developers. Furthermore, this supports the finding of sample projects for the evaluation since the corpus of possible projects is larger. The second criteria tries to make assumptions regarding the complexity of a linguistic architecture. Since a linguistic architecture is concerned with the usage of software languages and technologies, multiple languages could lead to a more interesting linguistic architecture which will finally provide a larger value to a developer. Furthermore, the approach is stressed by a complex example which should lead to more confidence regarding the usability of it.
Sawant and *Bacchelli* are providing a dataset that is describing API usage on GitHub. This paper is taken as a starting point. Their approach is scanning GitHub for projects that are using maven for dependency management. The pom files of those projects are used to extract the dependencies. This helps to build a list of the most popular APIs of maven projects on GitHub. To this long list the following criteria were applied by the authors to select API's for a more fine grained analysis:

- Reasonable code base of the API: more than 150 classes
- API is still maintained: more than 10 commits in a certain period of time

Five APIs are kept by the authors after the criteria have been applied. Since their criteria also make sense for this case study only these filtered APIs are investigated in depth for the selection. The result can be seen in Figure 3.

API & GitHub repo	Inception	Releases	Unique Entities	
			Classes	Methods
Guava google/guava	Apr 2010	18	379	2,010
Guice google/guice	Jun 2007	8	192	463
Spring spring-framework	Feb 2007	40	431	1,161
Hibernate hibernate/hibernate-orm	Nov 2008	77	585	1,963
EasyMock easymock/easymock	Feb 2006	14	10	86

Figure 3 Most used APIs of maven projects on GitHub according to [14]

The selected technologies are used frequently as the scan of GitHub proves and hence match the first criteria. The next step is to examine each of the proposed technologies and verify if they are also matching the second criteria. This is done in an informal way by looking at the tutorials of the technologies and identify if multiples (domain specific) languages appear. Further details about the technologies are collected to find reasons for the selection or exclusion of a technology. The following paragraph lists an explanation for the selection or exclusion of each technology. In the end, one should remain for the execution of the case study:

Guava is a collection of useful functions that facilitates best coding practices. In our opinion, it lacks the amount of different classes of artifacts. Furthermore, it focuses on the Java stack and lacks the use of different languages, e.g., XML. Hence, we do not consider this technology as input for this work.

Guice is used for dependency injection in Java projects. The use of annotations in this technology is very interesting and provides non trivial connections between artifacts. For this work we discard this technology because it is not using several languages in a sufficient way. Nevertheless, it is interesting for future work.

Hibernate uses languages like XML and Java including annotations. Furthermore, there are several kind of artifacts like mapping descriptions, configuration files and finally classes that are involved in performing the transactions. This technology is further interesting for the research community. For example *Nagy* et al. were doing research using Hibernate as a core technology. Hence, this technology fulfills all criteria.

Spring is an open source frameworks that has the goal to simplify development with JavaEE. It uses XML as well as Java annotations and got quite a few different artifacts like Java beans, XML files describing those beans and web content. On top of this, researchers like *Arthur* and *Azadegan* already made a case study concerned with the benefits of using it. Hence this technology fulfills all criteria. However, it does not have a correspondence relation of *MegaL* like Hibernate. Therefore, this technology will be moved to future work as well.

Easy Mock is a mocking framework for Java. It is widely used but as well lacks the use of several language. Furthermore, there are no artifacts with different roles since all are basically mocked objects. Hence this technology is too simple to extract a non trivial mega model .

Conclusion: The elaboration shows that Hibernate and Spring would be a good choose for the case study. We decided to focus our work on Hibernate since a lot Spring projects depend on Hibernate for object-relation mapping. In addition, Hibernate promises a more interesting linguistic architecture since a correspondence between an object and a table can appear.

5.1.2 Project Selection

The evaluation of our approach should take place by executing it on several projects available on GitHub. The selection of Hibernate for the case study was explained by the previous chapter. Since its usage was analyzed by *Sawant* and *Bacchelli* we get a dataset with all the maven projects on GitHub that were using Hibernate in 2015. This is taken as input to filter out the most interesting projects that are using Hibernate. Based on the following criteria the projects were selected:

- The top ten of the most used version of Hibernate should be taken into account. This condition helps to ensure that the version of Hibernate is relevant for several projects.
- The project should at least contain 200 classes. As a result, the project has a certain level of complexity and a non trivial linguistic architecture can be extracted.
- The GitHub project should have more than 10 contributors. This criteria ensures that the extracted linguistic architecture could be useful as a documentation since multiple developers are working on a project. Documentation is one of the possible concerns a linguistic architecture wants to address.

A list of projects that match all of these criteria are obtained with the help of SQL statements on the dataset. The selected projects are listed in Table 6 whereas the SQL statements that are applying the defined criteria to the dataset, provided by *Sawant* and *Bacchelli*, are listed in the Appendix Chapter 7.1.

Hibernate projects
Oscar
Druid
projectforge-webapp
IBPMiddleware
eeg-database
mateo
ROMS
oltpbench
frontlinesms-core
iTests-Framework

Table 6 Software projects that fulfill the criteria

5.2 Target Mega Model for Hibernate

The case study should automatically extract a mega model from the source code of the selected open source projects. The first step for this process is to define what exactly should be extracted. Therefore, this chapter gives an introduction to the linguistic architecture of a Hibernate use-case using the *MegaL* ontology. For the better understanding as a running example a *Patient* object is introduced. The mega model defined here should then be extracted with the help of custom rules that are based on the mega model. For simplification reasons the URI's of the entities and relations will not be shown in the Listings of this section. However, in the extracted mega model the triples are composed of URI's starting with *http://softlang.com*.

5.2.1 Basic Definitions

In a simple use case there exist at least two different *Systems*. One is Hibernate itself and the other one is the project to be documented. For simplification reasons both will not be discussed in detail with all their included *Artifacts*. The focus is rather on the artifacts that are playing a role for the mapping of a *Patient* object to a corresponding table. Such objects can be found in software systems as medical record databases and may look similar to Listing 10. For the actual mapping the Hibernate framework is used.

Listing 10 An exemplary patient class

```
1 package org.openmrs.db;
2
3 public class Patient extends Person {
4
5     public static final long serialVersionUID = 93123L;
6     private Integer patientId;
7     private String allergyStatus = Allergies.UNKNOWN;
8     private Set<PatientIdentifier> identifiers;
9 }
```

Java part: There exists a Java file in the project that contains the definition of the *Patient* class. In the context of the JVM, the object of this class is represented in a domain specific language called *PatientOLang*. Other objects would be represented by different OLangs. In fact, each Java class is defining its own OLang to which all objects of the class at run time are encoded in.
In the context of Java, classes are identified by the combination of their package and name. In the running example, we see that the *Patient* class can be identified by the string *org.openmrs.db.Patient*, which represents the concatenation of the package and class name. Since it follows the rules of a Java identifier, it is an *elementOf* the language *JavaClassUri*. The patient file itself declares an identifier that is composed of the package and class name and identifies the defined class. Hence in the context of *MegaL* this file defines the reference. For this scenario the new *MegaL* relation *decOccurs* was introduced. Its purpose is to describe that a specific identifier, e.g., a class or table name, is defined by a certain *Artifact*. Afterwards this element exits in the ecosystem of the project and can be referenced. Listing 11 shows the layout of the discussed content.

Listing 11 Definition of mappable objects in MegaL

```
1 patient.java elementOf Java
2 patient.java defines patientOLang
3 org.openmrs.db.Patient elementOf JavaClassURI
4 patient.java decOccurs org.openmrs.db.Patient
```

SQL part: The other side of the mapping is the relational model that is defining the table to which the object is mapped. In the running example, a *Patient* object will be mapped to a table called *PatientTable*. The table itself has to be created. There are several ways to create a table but in this scenario it is assumed that the table is build by a SQL create table statement like Listing 12 shows. This SQL command is part of a SQL file. In fact, it is just the fragment of the SQL file that is containing exactly the range of the create statement. The range is defined as the start and end position.
This fragment is defining the relational language of the table which is basically the representation of the table in the relational database. The exact layout of the language depends on the relational database in use. Hence, it is kind of an abstraction of a concrete table.
After the create statement was executed a table can be referred to with its table name. Therefore, the actual table name, which is a *QualifiedName*, is declared by the create statement. The newly introduced relation *decOccurs* of *MegaL* is used in this scenario as well. The described content is encoded in Listing 13 in *MegaL* syntax.
In other scenarios, the table can be created by technologies like for example Liquibase[4] or Hibernate is creating the schema itself.

[4]http://www.liquibase.org/

Listing 12 An exemplary create statement for a patient table

```
CREATE TABLE patientTable (
  ID INT NOT NULL auto_increment,
  Patient_Id VARCHAR(255) NOT NULL,
  AllergyStatus TEXT NOT NULL,
  FName TEXT NOT NULL,
  LName TEXT NOT NULL,
  BirthDate DATE NOT NULL
);
```

Listing 13 Definition of the relational model in MegaL

```
tables.sql elementOf SQL
tables.sql#0:214 type Fragment
tables.sql#0:214 partOf tables.sql
tables.sql#0:214 elementOf SQLiteCreateTableStmt
tables.sql#0:214 decOccurs patientTable
tables.sql#0:214 defines patientRLang
patientTable elementOf QualifiedName
```

Final relation In the end, it has to be identified to what fragment of the SQL file the Java file is corresponding to (Listing 14).

Listing 14 Relation of interest

```
patientFile correspondsTo tables.sql#12:30
```

The mapping function: Hibernate is basically translating the *PatientOlang* to the *PatientRlang* and vice verse. For simplification reasons the translation from the relational to the object model is ignored. Eventually, it will be the same, only the input and output are switched. Since *MegaL* is normally not encoded into *RDF triples* the two relations *input* and *output* are introduced as properties in RDF. That way the function usage can be encoded into *RDF*.
The translation itself can be described as a function that is transforming one language to the other language. This function is different for each object and table pair but is still a *partOf* the overall *OR-Mapping* function. A suggested *MegaL* layout is presented in Listing 15.

Listing 15 Definition of the specific mapping function

```
patient−ORMapping type Function
patientOLang input patient−ORMapping
patient−ORMapping output patientRLang
patient−ORMapping partOf OR−Mapping.
```

How the object is mapped to the table is defined by a mapping description which is either defined by a XML file or Java annotations. Both will be discussed separately.

5.2.2 XML Mapping

The mapping itself is described by the Patient-OR-Mapping XML file. In the context of the *MegaL* ontology the mapping is represented as a function. Consequently, the XML file defines a corresponding mapping function.
For the description of the mapping, the actual Java class that should be mapped as well as the table it is mapped to has to be defined. Therefore, the file is pointing to both references. This is described by the newly introduced *MegaL* relation called *refOccurs*. This one is used when a file is using a reference. Usually this reference should be defined with the help of the *decOccurs* relation by another file. An exemplary layout of such a XML mapping file is shown in Listing 16. This file does *conformTo* a specific XSD-schema. Therefore, Hibernate is able to read its content and to extract the mapping information. Listing 17 shows such a definition in *MegaL*.

Listing 16 An example of a Hibernate mapping description

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >

<hibernate-mapping package="org.openmrs.db">
   <class name="Patient" table="PatientTable" mutable="false">
   <property name = "allergyStatus" column = "AllergyStatus" type = "string"/>

   .....
   </class>
</hibernate-mapping>
```

Listing 17 Definition of the object specific mapping function

```
    patient.hbm.xml defines patient-ORMapping
    patient.hbm.xml refOccurs patientTable
    patient.hbm.xml refOccurs org.openmrs.core.Patient
    patient.hbm.xml conformsTo hibernate-mapping.xsd
```

5.2.3 Hibernate annotations

Another possibility to describe the mapping between an object and a relational table is to annotate the Java class with information regarding the mapping. An exemplary outline of such a class definition can be seen in Listing 18. In that scenario, a separate XML file does not exist but the information are rather encoded in the *patientFile*. In fact, the *patientFile* file is containing a fragment which is an element of the *JavaAnnotatedElement* language. This fragment is defining the mapping function and hence basically plays the role of the previous mentioned mapping XML file. The resulting *MegaL* model can be seen in Listing 19.

Listing 18 The annotated patient class

```
1  package org.openmrs.db;
2
3  import javax.persistence.*
4
5  @Entity
6  @Table(name="patientTable")
7  public class Patient extends Person{
8
9
10 }
```

Listing 19 Definition of the specific mapping function with Java annotations

```
1      patient.java#3:5 type Fragment
2      patient.java#3:5 partOf patient.java
3      patient.java#3:5 elementOf JavaAnnotatedElement
4      patient.java#3:5 refOccurs patientTable
5      patient.java#4:5 defines Patient−ORMapping
```

5.2.4 Service definition

The earlier defined mapping function should be executed by a Java file. Otherwise the objects would never be persisted into the database and the mapping actually never took place. The execution happens when a related Hibernate function, like for example *saveOrUpdate*, is called with a persistable object as argument. An example for such a function usage can be seen in Listing 20. The service itself is using the Java class in question as well as a Hibernate package which enables the system to execute a function that performs the mapping. There are several Hibernate packages and methods that are capable of performing the mapping but with slightly different implementations and results. The description of this scenario in *MegaL* is shown in Listing 21.

Listing 20 Example of a service which is executing a patient mapping function

```
1  import org.openmrs.db.Patient;
2  import org.hibernate.SessionFactory;
3  import org.hibernate.cfg.Configuration;
4
5  public class HibernatePatientDAO {
6      private SessionFactory sessionFactory;
7
8      public HibernatePatientDAO(Configuration config) {
9          sessionFactory = config.configure().buildSessionFactory()
10     }
11
12     public Patient savePatient(Patient patient) throws DAOException {
13             if (patient.getPatientId() == null) {
14                 sessionFactory.getCurrentSession().saveOrUpdate(patient);
```

```
15            }
16            return patient;
17    }
18 }
```

Listing 21 The service data encoded into MegaL

```
1    hibernatePatientDAO.java elementOf Java
2    hibernatePatientDAO.java refOccurs org.openmrs.core.Patient
3    hibernatePatientDAO.java refOccurs org.hibernate.Session
4    hibernatePatientDAO.java uses patient−ORMapping
```

5.2.5 Role definition

Three of the artifacts have a special purpose in the Hibernate ecosystem. For these, roles are introduced. For example a mapping file has the specific role *HibernateMapping*. The object to be persisted in the relational database has the role *HibernatePersistedObject*. Last but not least, does the Java file, that executed the mapping, has the role of a *HibernateService*. Listing 22 shows the roles manifested in the running example.

Listing 22 The different roles found by using the Hibernate technology

```
1    hibernatePatientDAO hasRole HibernateService
2    patient.java hasRole HibernatePersistedObject
3    patient.hbm.xml hasRole HibernateMapping
4    patient.java#3:5 hasRole HibernateMapping
```

5.3 Iterative Rule Development

The goal of the approach is to extract the mega model that was defined in Chapter 5.2. To extract that model several rules have to be developed which will extract the necessary relations of the mega model for an arbitrary project. Those rules were created in an iterative matter. The most basic knowledge was extracted first and then step by step a in depth linguistic architecture was created. The execution of the rules by *Jena* does not have to be in the same order but due to the dependencies between the triples it will be close to it. Each of that creation steps are shown in Figure 4. At the end of the process all the relations of the mega model should be extracted in form of *RDF triples*. All of the rules including the source code for the built-ins are available on GitHub[5].

Figure 4 The extraction process used by our approach

5.3.1 Language detection

The first step for the extraction of the MegaL model is to identify the languages of the artifacts in the software system. In the context of the Hibernate mega model just a few languages are used. This approach aims to just identify the artifacts that are an *elementOf* these languages. Namely these languages are *Java*, *SQL*, *XML*, *DTD* and *XSD*. In addition, the domain specific language *SQLCreateStmt*, a subset of *SQL* that covers all the create table statements, is taken into account.

For the identification of the language of an artifact two approaches are at hand. On the one hand, the suffix of the file gives an indication about the language the file is of. On the other hand, parsing a file would identify the language without any doubt. Our goal is to stick with the safe method of parsing as long as it is feasible. In the case of DTD and SQL it is very difficult to create a working parser. This is the case for *SQL* since it manifests in a lot different dialects. A SQL statement that is valid for a MySQL database is may not valid for an Oracle database. Hence, it was decided to use parsing for *SQLCreateStmt* commands, which represent a create statement. For DTD the suffix identification was chosen since it was not possible to find a stable parser. A summary of the methods used for each language can be found at Table 7. If both are selected then the suffix is used to build a hypothesis which is then verified by parsing it.

[5]https://github.com/fruether/RuleEngine

Table 7 Shows the methods used to identify a language

Language	Parsed	Suffix
SQL		x
Java	x	x
XML	x	x
XSD	x	x
DTD		x
SQLCreateStmt	x	x

Artifact based language detection:
In the *model creation phase* all the existing *Artifacts* of the project were already loaded as triples to an initial model. The Jena reasoner is not powerful enough to identify a language. Therefore, the *Parse* built-in was coded in Java. This built-in will load the content of a file and tries to parse it accordingly to its suffix. This means that a file with the suffix *.java* will be parsed by a Java parser. In case the parser was successful the built-in returns true and binds the variable to an URI identifying the language Java. This works the same way for *XML* and *XSD*. The complete rule is presented by Listing 23.

Listing 23 Identification of the language of an artifact

```
[LanguageDetection:
    (?file rdf:type Artefact)
    Parse(?file, ?language)
    ->
    (?file sl:elementOf ?language)
]
```

Fragment based language detection:
The identification of the domain specific language *SQLCreateStmt*, which represents a create table statement in SQL, can't be established by the previous introduced rule. This is the case because a *Fragment* of a file is an element of this language. For instance imagine a large *SQL* file that is composed of a create and multiple insert statements. The latter ones are populating the previous created table. In this scenario, only the fragment of the file containing the create statement is of the type *SQLCreateStmt*.
Yet we don't want to scan every file for a create statement. Therefore, first all the artifacts ending with *.sql* are identified. This is archived by the previous introduced rule, which assigns those files the language *SQL*.
The extraction of the create statement is handled in the following way. First of all, a regular expression extracts the create statement out of the file. The result is saved as a fragment *Entity* that is identified by an URI composed of the file path, the start and end position of it. Afterwards, this fragment is parsed with ANTLR[6] and it is checked if it really is a syntactically correct SQL create statement. Both steps are performed by a specific built-in called *CreateStmtExtraction* that expects a SQL file as input. It is a head

[6] http://www.antlr.org/

built-in since it adds all the necessary triples to the resulting model. Listing 24 shows the complete rule. The result of the rule is illustrated in Listing 25 where the variable *?statement* is the fragment containing the URI that identifies the SQL create statement. The extraction of the language *JavaAnnoatedElement* is similar but will be elaborated in Chapter 5.3.3

Listing 24 Fragment extraction and parsing for a SQL create statements

```
[ExtractAndParseCreateStmts:
    (?file sl:elementOf sl:Language/SQL)
    ->
    CreateStmtExtraction(?file)
]
```

Listing 25 Result of the rule shown in Listing 24

```
    (?statement rdfs:type sl:Fragment)
    (?statement sl:partOf ?file)
    (?statement sl:elementOf sl:Language/SQLCreateStmt)
```

5.3.2 Basic language based extractions

Before it is possible to extract all the Hibernate specific triples further basic extractions based on the used *Language* have to take place. The languages of the files are known since the triples using the property *elementOf* have been extracted by the previous rule. Based on this, further steps can be performed to build knowledge about the system. For example it is useful to know what schema a XML file is following. Another tough problem is that in the context of Java a class is not identified by the file path but by the canonical class name[7], which is a combination of the package and class name.

ConformsTo extraction: XML files could follow a schema. Such a schema can be defined either in *XSD* or in *DTD*. Since a project is maybe using *XML* files to describe the mapping of an object to a table, it would be of interest to identify the conformation between *XML* and Hibernate *XSD* files. A built-in called *XSDCheck* was implemented in Java that is used in the body of a rule and returns true if a XML file is following the requested XSD schema. The focus is mainly to find artifacts with the fingerprint of a technology like Hibernate. Hence both files should be found in different *Systems*. Listing 26 shows the rule that helps to achieve this.

Listing 26 Checking if a XML file conforms to a XSD file

```
[XSDCheck:
    (?a sl:elementOf sl:Language/XML)
    (?b sl:elementOf sl:Language/XSD)
    (?a sl:partOf ?system1)
    (?b sl:partOf ?system2)
    notEqual(?system1, ?system2)
```

[7] https://docs.oracle.com/javase/8/docs/api/java/lang/Class.htmlgetCanonicalName–

```
7       XSDCheck(?a,?b)
8       −>
9       (?a sl:conformsTo ?b)
10  ]
```

Class extraction: In the context of Java, a class is identified by its canonical class name, which is composed of the package and class name. It is important to know what Java file is declaring which class since a Hibernate XML mapping file is referencing the class and not the file, e.g., it contains *org.package.Class* and not */org/package/Class.java*.
Therefore, one of the early steps is to extract what files declare (*decOccurs*) which class. The class is identified with the help of the built-in *RetrieveClass* that takes a Java file as argument, parses it and returns the combination of package and class name as an URI. The complete rule is presented in Listing 27.

Listing 27 Extraction of the Java class URI

```
1   [JavaClassIdentification:
2       (?file sl:elementOf sl:Language/Java)
3       RetrieveClass(?file, ?classname)
4       −>
5       (?file sl:decOccurs ?classname)
6       (?classname sl:elementOf sl:Language/JavaClassURI)
7   ]
```

5.3.3 Hibernate mapping identification

XML Mapping discovery: One possibility to map an object to a relational table with the Hibernate framework is to use a XML file to describe the mapping. This mapping file *conformsTo* to a Hibernate specific XSD format. Those kind of triples were already extracted in the previous phase.
MegaL introduces the entity *Role*, which describes the purpose of a file. The earlier described file would have the role *HibernateMapping*. The built-in *HibernateRoleIdentification*, usage shown in Listing 28, assigns to each XML file that conforms to a Hibernate XSD file a valid role. The built-in checks the file name of the schema to identify the role.

Listing 28 Hibernate role identification with the help of a built-in

```
1   [HibernateRoleAssignment:
2       (?file sl:elementOf sl:Language/XML)
3       (?file sl:conformsTo ?schema)
4       (?schema sl:partOf sl:hibernate)
5       HibernateRoleIdentification(?schema, ?role)
6       −>
7       (?file sl:hasRole ?role)
8   ]
```

After the roles have been identified, the table and class that are referenced in the XML file can be extracted. The obtained information will be translated to an URI and added

to the model.
The rules for the extraction are using two built-ins to establish their goals. One is called *HibernateMappingGetClassURI* and its purpose is to extract the referenced class in the Hibernate mapping file. This is possible because thanks to the schema defined by Hibernate it is known with what XPath the reference to the class could be retrieved. Therefore, the built-in is simply parsing the XML file and asking for the attribute by its id. This process is called for each *HibernateMapping* XML file and linked to the extracted class URI by the *refOccurs* relation. Since it is known by the previous defined rule (Listing 27) which Java file is declaring this class a chain between both files exists.
The mapping file is not just referencing the mapped class but the table it should be mapped to as well. Hence the similar built-in *HibernateMappingGetTable* is called to extract the table out of the XML file. This table is identified by an URI.
A SQL file may be declaring that table name by using a create statement. Therefore, this URI is maybe linked to a SQL file by the relation *decOccurs*. Since a table name is just a *QualifiedName* it can under circumstances not be identified as a table or another *QualifiedName* is by error seen as a table name. Therefore, a sub-property of *refOccurs* was introuced that is specifying that in fact a *HibernateMapping* file is referencing a relational table name. The name of this relation is *HibernateMappingRefOccursRelationalTable*. A sub-property in *RDF* is doing the same as an inheritance of object oriented programming.

Listing 29 Hibernate role identification with the help of a built-in

```
1  [HibernateMapping:
2      (?file1 sl:elementOf sl:Language/XML)
3      (?file1 sl:hasRole sl:HibernateMapping)
4      HibernateMappingGetClassURI(?file1, ?class)
5      ->
6      (?file1 sl:refOccurs ?class)
7  ]
8
9  [HibernateMappingReferencedTables:
10     (?file sl:hasRole sl:HibernateMapping)
11     (?file sl:elementOf sl:Language/XML)
12     HibernateMappingGetTable(?file, ?table)
13     ->
14     (?file sl:HibernateMappingRefOccursRelationalTable ?table)
15 ]
```

Annotation discovery:
The extracted MegaL model contains a fragment which represents, a with Hibernate elements annotated, Java class. This fragment is as well defining an O/R-Mapping. In order to extract the same relations as for the XML mapping file a new rule has to be created. The rule is using a built-in called *HibernateAnnotationExtraction*, which is located in the head and is extracting the table name that is referenced by the annotation. Of course the extraction only takes place if the class was annotated with one of the valid Hibernate annotations. Otherwise nothing is added to the model. This built-in is just

called on files that match certain pre conditions. First of all, the file has to be an element of the language *Java*. Secondly, the file has to import the package that is defining the annotations. Those annotations are defined in the package *"javax.persistence"*. Therefore, a separate body built-in checks if it is included. Only if both conditions are met a scan for the Hibernate annotations is taking place. After the rule is executed the fragment is extracted, it was assigned as an element of *JavaAnnotaedElement*, the roles are set and a reference to the table is manifested. Listing 30 shows the complete rule and Listing 31 illustrates the result of the successful execution of the head built-in. However, the variables *?fragment* and *?table* would be concrete URIs.

Listing 30 Mapping definitions with the help of annotations

```
[CheckAnnotationHibernate:
    (?file sl:elementOf sl:Language/Java)
    CheckLiteralImported(?file, "javax.persistence")
    ->
    HibernateAnnotationExtraction(?file)
]
```

Listing 31 The result of the execution of the rule shown in Listing 30

```
    (?fragment rdfs:type sl:Fragment)
    (?fragment sl:partOf ?file)
    (?fragment sl:elementOf sl:Language/JavaAnnotatedElement)
    (?fragment sl:HibernateMappingRefOccursRelationalTable ?table)
    (?fragment sl:hasRole sl:Role/HibernateMapping)
```

5.3.4 Language definitions

The required MegaL model shows that there exist two specific languages that are defined by a Java or SQL file. Those languages represent the object and relational table during runtime. Those languages will not be defined for each Java and SQL file but only for the ones that are relevant for the mapping process. The rule is similar for the definition of the *RLang* and *OLang*. Only the definition of the *OLang* will be discussed; the other rule is analogue except that the fragment of the create statement is defining the language based on the table name.
Basically the *HibernateMapping* file, which is identified by its role, is used to retrieve the class that should be mapped. When the class is known, the file itself can be derived since it *decOccurs* the referenced class name. Finally, the language can be defined based on the class name. For the extraction of the class name the *GetClassLiteral* built-in is used. The extracted language is enriched with additional information in the conclusion, for example that it is a subset of the language *JVMObject*. Listing 32 is showing the rule that is defining the specific *OLang*. A similar rule exists for the *RLang* definition and for the *OLang* definition, that is based on a XML mapping description.

Listing 32 Definition of the OLang for each annotated class

```
[HibernateObjectDefinitionJavaMapping:
    (?fragment sl:hasRole sl:HibernateMapping)
    (?fragment sl:elementOf sl:Language/JavaAnnotatedElement)
    (?fragment sl:partOf ?file)
    (?file sl:decOccurs ?class)
    (?class sl:elementOf sl:Language/JavaClassURI)
    GetClassLiteral(?class, ?classLanguage)
    −>
    (?file sl:defines ?classLanguage)
    (?classLanguage rdf:type sl:Language)
    (?classLanguage sl:subsetOf sl:JVMObject)
    (?file sl:hasRole technologies:Hibernate/HibernatePersistedObject)
]
```

5.3.5 Mapping function definition

The files that are describing the O/R-Mapping of an object with the help of the Hibernate framework are defining specific mapping functions. These functions are responsible for mapping the objects to the corresponding tables. A name for each of those functions has to be created. Therefore, for each file with the role *HibernateMapping* a built-in is used to create a valid function name, which is based on the class that should be mapped, e.g., a *Patient* class mapping would lead to the function name *Patient-ORFunction*. Two different rules are needed to handle on the one hand the Java annotations and on the other hand the XML description. The reason for this is, that in one case a file and in the other case a fragment is defining the specific mapping function. Both rules are laid out in Listing 33.

Listing 33 Definition of the mapping function

```
[HibernateMappingFunctionDefinitionXML:
    (?file sl:hasRole sl:HibernateMapping)
    HibernateGetMappingType(?file, ?function)
    −>
    (?file sl:defines ?function)
    (?function rdf:type sl:Function)
    (?function sl:partOf sl:OR−Mapping)
]

[HibernateMappingFunctionDefinitionJava:
    (?fragment sl:hasRole sl:HibernateMapping)
    (?fragment sl:elementOf sl:Language/JavaAnnotatedElement)
    (?fragment sl:partOf ?file)
    (?file sl:elementOf sl:Language/Java)
    HibernateGetMappingType(?file, ?function)
    −>
    (?fragment sl:defines ?function)
    (?function rdf:type sl:Function)
```

```
19      (?function sl:partOf sl:OR−Mapping)
20  ]
```

5.3.6 Input and output of function detection

After the function triples for a specific object mapping were added to the model input and output values have to be assigned to this function. The input is in this case the object language that is defined by the Java file. Whereas, the generated output is the relational language that is defined for example by a SQL create statement.

Listing 34 The linking of the function to its input and output

```
1  [HibernateMappingFunctionJavaInputDefinition:
2      (?fragment sl:hasRole sl:HibernateMapping)
3      (?fragment sl:elementOf sl:Language/JavaAnnotatedElement)
4      (?fragment sl:partOf ?file)
5      (?fragment sl:defines ?function)
6      (?function sl:partOf sl:OR−Mapping)
7      (?file sl:defines ?language)
8      −>
9      (?language sl:input ?function)
10 ]
11
12 [HibernateMappingFunctionSQLOutputDefinition:
13     (?file sl:hasRole sl:HibernateMapping)
14     (?file sl:defines ?function)
15     (?file sl:HibernateMappingRefersToRelationalTable ?table)
16     (?table sl:partOf ?sqlFragment)
17     (?sqlFragment sl:defines ?relationalLanguage)
18     −>
19     (?function sl:output ?relationalLanguage)
20 ]
```

Correspondence:
The correspondence between the SQL fragment and the Java file is now easy to retrieve. The files that are linked by the mapping function in an indirect way need to be linked by an explicit relation called *correspondsTo*.

5.3.7 Usage detection

Another interesting fact that should be extracted according to the required MegaL model is what files are using the mapping functions that have been defined by the previous rules. The identification is not trivial since under normal circumstances a type system, which is identifying the exact types used in a function call, would be necessary. This would be the only way to guarantee that for example the *saveOrUpdate* function of Hibernate is in fact called with the object in question. Manifesting a type system is out of scope for now since it would require the compilation of the projects. Therefore, another

may less precise heuristic is used.
For an object to be mapped either way the minimal condition that has to hold is that it has to be declared somewhere in the service file. This condition can be checked without a type system since the included packages, the declared objects and the complete canonical names of the persistable classes in the Java ecosystem are known.
To summarize, the first condition is that a class is declared, which is connected with the *decOccurs* relation to a Java file that has the role *JavaPersistedObject*. A rule called *HibernateImportUsedDetection* (see Listing 35) is responsible for checking this condition. For performance optimization this is not checked for every file but only for the ones that are using a Hibernate related package. Its performance intensive due to the built-in *CheckClassReference* which is parsing each file to extract the declaration. The aim is to just do this for the files that have a high chance of using such a mapping function.
The second condition is that the file is executing a function that indicates that the mapping takes place. This property is as well used by the implementation of the first rule. The first thing to check for this approach is that one of the packages is imported and used which implements the methods that are required for performing the mapping. The latter information is as well extracted by the built-in *CheckClassReference*. A list of valid packages is existing in the Chapter 7.3 of the Appendix. In addition, it is secondly checked if one of the methods, that are actually performing the mapping, is called in the file. This condition is supported by the built-in *CheckHibernateMethodUsage*, which is simply verifying that a function with one of the names in the appendix is called. Just using the name is not type safe and hence wrong results can appear. Therefore, it is just a heuristic.

Finally, *HibernateServiceDiscovery* is the rule that kicks in after the object and package usage describing *RDF-triples* are extracted. Now the information just have to be gathered to create the *use* relation between a file and the mapping function. However, it should be kept in mind that the approach uses a weak heuristic so that a lot false positive cases could get extracted.

Listing 35 The linking of the function to its input and output

```
1  [ImportDetection:
2      (?file sl:elementOf sl:Language/Java)
3      (?package rdf:type sl:Package)
4      CheckClassReference(?file, ?package)
5      CheckHibernateMethodUsage(?file)
6      ->
7      (?file sl:uses ?package)
8  ]
9
10 [HibernatePersistedObjectUsedDetection:
11     (?file1 sl:elementOf sl:Language/Java)
12     (?package sl:partOf sl:hibernate)
13     (?file1 sl:uses ?package)
14     (?file2 sl:hasRole technologies:Hibernate/HibernatePersistedObject)
15     notEqual(?file1, ?file2)
```

```
16      (?file2 sl:decOccurs ?class2)
17      (?class2 sl:elementOf sl:Language/JavaClassURI)
18      CheckClassReference(?file1, ?class2)
19      ->
20      (?file1 sl:uses ?class2)
21  ]
22
23  [HibernateServiceDiscovery:
24      (?file sl:elementOf sl:Language/Java)
25      (?file sl:uses ?package)
26      (?package sl:partOf sl:hibernate)
27      (?file sl:uses ?class)
28      (?filePers sl:decOccurs ?class)
29      (?filePers sl:hasRole technologies:Hibernate/HibernatePersistedObject)
30      (?filePers sl:defines ?language)
31      (?language sl:input ?function)
32      (?function sl:partOf sl:OR-Mapping)
33      ->
34      (?file sl:uses ?function)
35      (?file sl:hasRole technologies:Hibernate/HibernateService)
36  ]
```

5.3.8 Rules for the support of special cases

Hibernate entity is using another entity: In the context of Hibernate, it can always be the case that an object is using another object. For example the class *Patient* could have an object of type *Address* as an attribute, which is itself composed of several strings that store the city, address and country. Both are perhaps entities of Hibernate[8]. It is not always necessary to have a specific service which maps the objects of the class *Address* to a table. This operations can as well be handled by the *PatientService*, which maps all the *Patient* objects to a table. Therefore, a rule is created that states that every time a persisted object is using another persisted object, one of the mapping functions is using the other mapping function. This is the case since there exist two different types of objects to map. In the above scenario both would be mapped by the *PatientService*. It would make no sense to have a *AddressService* since the address object just saves data and would not exist without an object that uses it.

Parent class is using a package: One of the rules shown in Listing 35 is checking if a certain package was used. However, there exist cases where a class inherits from a super class. The super class is itself importing the package in question but not the child class. The child class is using a class, which should be persisted, though. In that scenario the importing of the package and the declaration of the object are separated. The rule shown in Listing 36 is used to deal with that special case. In fact, each class that inherits from a class will use the same packages as the parent class does. An exception exists if *javax.persistence* is imported since then the inheritance is most likely because it is a

[8] https://docs.jboss.org/hibernate/annotations/3.5/reference/en/html/entity.html#entity-hibspec-entity

persisted object. The *CheckExtension* built-in simply validates if a class is extended by another class declared in *file2*.

Listing 36 Detection of inheritance between different persistable objects

```
[ImportedByExtensionDetection:
    (?package rdf:type sl:Package)
    (?file sl:uses ?package)
    (?file sl:decOccurs ?class)
    (?file2 sl:elementOf sl:Language/Java)
    notEqual(?file, ?file2)
    CheckNotImported(?file2, "javax.persistence")
    CheckExtension(?file2, ?class)
    ->
    (?file2 sl:uses ?package)
]
```

5.3.9 Implementation details

So far just built-ins have been introduced. However, in the architecture of the system several services are implemented that can be used by the built-ins and enable further functionality. For example a *FileRetrievementService* exists which translates an URI to the actual file path and further provides functionality to read a file from the disk. On top of this, several services exist that deal with language specific functionality. As an example, there exists a *JavaService* that provides functionality related to the parsing of a Java file. The imported packages as well as the declared objects can be retrieved with the help of methods that are implemented in it. This service is used by several built-ins and since just one instance exists the parse tree can be cached and all the Java related functionality is stored in one file, which is coherent. Figure 5 illustrates the layout. As it can be seen, multiple rules are perhaps calling the same built-ins or a rule itself is calling several built-ins. Nevertheless, the built-ins depend for a lot of their functionality on general services that are independent of the Jena infrastructure. Multiple built-ins can use the same *Services*. Therefore, routines that have to be used by multiple built-ins should be aggregated into a coherent service. A list of all the services that can be used by the built-ins can be found in Table 8.

Name	Purpose
FileRetrievementService	Manages the access of files on the file system. Especially reading the content of a file and eventually caching it.
LanguageService	This services is concerned with the identification of the language of a file. Hence it is itself using other more language specific services like the *JavaService*. However, it provides a single interface for the identification of a language.

JavaService	The Java service encapsulates the functionality related to parsing a Java file, e.g., getting the declared object, methods or used packages.
SQLService	Encapsulates SQL specific functions for example parsing a statement, extracting the create statement or returning the table name.
ValidationService	This one is used to validate that a file is conforming to a schema. All the functions related to this can be found in this service.
URIService	This service is mainly concerned with storing the names of relation and concepts of the Ontology and combining it with the URIs that are supposed to be used. This service helps to avoid typos in an URI and hence destroying an existing chain.

Table 8 Services that can be used by the built-ins

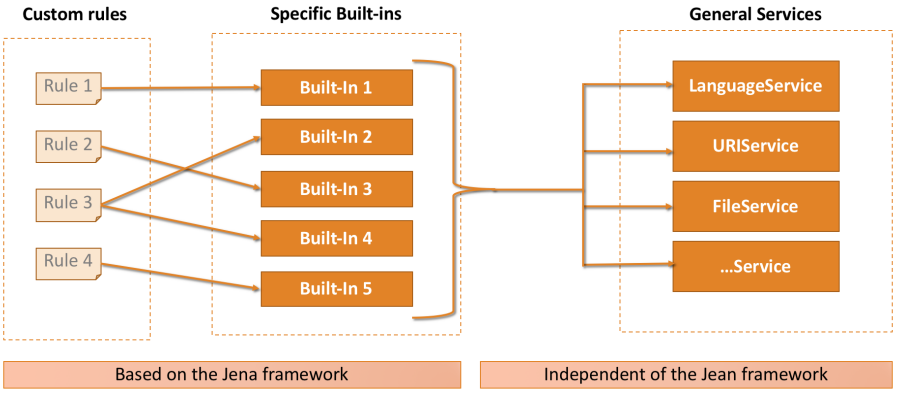


Figure 5 The relation between rules, built-ins and services

5.4 Evaluation

After the rules are developed they can be evaluated with open source projects. A sample of projects has been selected by the methodology described in chapter 5.1.2. The next step is to execute the process on these projects. The evaluation is based on the extraction of different metrics that summarize the result. These metrics are examined by two fundamentally different approaches that are called internal and external evaluation. The difference between both is that the internal is just looking at the retrieved data set whereas the external is comparing it to another data source:
(1) The internal approach checks if the previous defined constraints are holding. These constraint are build with help of the introduced metrics and should ensure that the (internal) result itself is consistent. In fact, the constraints should reflect the result that is expected by the target mega model. With the help of the constraints the approach validates if the retrieved result has any contradictions. A break of the constraint has to be investigated. In case it is broken because the state of the project is reflecting this, it is acceptable. Otherwise, our rules would contain an error that has to be fixed. Therefore, for each break a reason has to be found.
(2) The external approach is validating the result against a different (external) data set. This data set is provided by an oracle that is collecting the characteristics of the project with a different method. Both data sets will be compared and an explanation for the derivation has to be found. The found explanation should explore which of both classification were correct and finally show how good the developed approach performs compared to an oracle. In an optimal scenario both approaches should calculate the same result.

5.4.1 Metrics

The evaluation approach depends on metrics that summarize the extracted state of the system. This paragraph aims to introduce these metrics and explains why they are necessary. For the better understanding they are classified into different logical categories.
Mapping definitions: The most basic metric that provides a value is to count the *HibernateMapping* files that are identified. Furthermore, this metric introduces two sub metrics which count how often the mapping was described with XML and annotations. This metric is important because based on the mapping definitions further details are extracted. Therefore, it is critical to get an overview about the number of mappings that are detected.
Mapping functions: The mapping files itself introduce mapping functions. Those functions are taking as input and output specific languages called *RLang* and *OLang*. Another metric counts how many of those languages exist that are connected to a mapping function. That way missing input or output can be detected. This helps to validate if a mapping function is well formed.
The mapping of the objects to the table is not performed automatically. At some point those specific mapping functions should be used. Hence, each *use* relation that has a Hibernate function as object increases a certain metric which counts how many func-

tions are used. In addition, it is counted how many different Java files are using this functions. This enables to detect functions that are never executed.

Java and SQL relations: Overall there exist several relations between the SQL tables, Java files and the mapping descriptions. In fact, a Hibernate mapping file is referencing a table. How often this happens is stored by the metric *HibernateMappingRefersToRelationalTable*. In the best case both files that are referenced by the Hibernate mapping file do *correspondsTo* each other. Every correspondence is counted by the metric *HibernateCorrespondence*. This metric can be distinguished between XML and annotation based correspondence.

So far the metrics focused on the Java part of the software project. However, the SQL files that are contained in the project are as well containing details regarding the design of the project. The metric *TableStatement* stores the number of tables in the project that are created with the SQL create table statement. This enables the developer to compare the amount of tables in the project with the number of persistable classes.

Final remarks: It should be mentioned that some metrics exist in two versions, e.g., with and without the suffix *Distinct*. This suffix defines that each element is just counted ones and not multiple times. For example without the *Distinct* ending each create statement with a table would be counted. With the according suffix each table name would just be counted ones even when it is created by multiple statements.

A complete list of the defined and extracted metrics is presented in Table 9.

Name	Description
HibernateMapping	This metric counts how many files exist with the role *HibernateMapping*
HibernateMappingJava	Counts the amount of files that are an *elementOf* Java and further having the role *HibernateMapping*
HibernateMappingXML	Counts the amount of files that are an *elementOf* XML and further having the role *HibernateMapping*
ImplementedFunctions	This metric simply counts how often a function that is *partOf* Hibernate is *used* by another artifact.
ImplementedFunctions Distinct	This metric simply counts how often a function that is *partOf* Hibernate is *used* by another artifact. However, each function with the same name is just counted once.
FunctionUsingFiles	This metrics counts how many files are in fact using a mapping function.
MappingFunctionWith OLang	This metric counts the amount of *OLangs* that are the input of a function.
MappingFunctionWith RLang	This metric counts the amount of *RLangs* that are the input of a function.
HibernateMappingRefers ToRelationalTable	This metric counts how often there exists a reference to a SQL table in a HibernateMapping file.

HibernateMappingRefers ToRelationalTableJava	This metric counts how often there exists a reference to a SQL table in a HibernateMapping file which is *elementOf* a Java file.
HibernateMappingRefers ToRelationalTableXML	This metric counts how often there exists a reference to a SQL table in a HibernateMapping file which is *elementOf* a XML file.
HibernateCorrespondence	This metric counts how often a *correspondsTo* relation exist between a XML and a Java file.
ReferencesToTable	This element counts how many tables have been created by SQL files.

Table 9 The developed metrics for the evaluation

5.4.2 Internal Evaluation

One way for the evaluation of the approach is to look at the result of the analysis and to check if it is coherent or if there are any contradictions. If latter is the case they are investigated in detail.
This approach does not discover missed relations but it should point out special properties of the system at hand. This is the case since broken constraints may hint a bug or design flaw in the application.
To perform such an evaluation different constraints will be set based on the previously introduced metrics. Those constraints are expected to hold in any project if there does not seem to be a contradiction or unexpected result. A list for acceptable reasons for the break of a constraint can be found in Chapter 5.4.2. If there is no way to explain a break by the properties of the project then our rules contain a bug. Therefore, it is necessary that a valid reason is found for each constraint that is broken.
The approach just looks at the metrics and the project files. No other results provided by a separate source or tool are necessary to execute it. Therefore, it is called internal validation.

Constraints:

(1) Checking for well defined functions: The previously defined metric *HibernateMapping* counts how often a mapping definition exist. It should always be the case that exactly the same number of *RLangs* and *OLangs* exist in the project as mapping definitions. Otherwise a mapping function without input or output would exist. If this is not the case either a table or an object will not be defined. This should not be the case.
(2) Correspondence restrictions: Different files may correspond to each other. Since in the best case each *HibernateMapping* leads to at least one *correspondsTo* relation, the number of this relations should never be lower than the amount of *HibernateMappingFiles*. It is at least one because a table can be created by different SQL statements. This is the case because different supported SQL vendors, like Oracle or Microsoft, may have a separate *init.sql* file in the software project. As a result, there would be a correspondence

between a Java object and each of those SQL fragments.
(3) Function usage: Each mapping function in the Hibernate context should be executed at least ones. Therefore, a reasonable constraint is to say that the metric *Implemented-FunctionsDistinct* should be equal to *HibernateMapping*. Otherwise some functions seem to be never called which would result in an odd state of the system at hand.
Summary:
$(1) HibernateMapping = RLangs = OLangs$
$(2) HibernateMapping <= HibernateCorrespondence$
$(3) HibernateMapping = ImplementedFunctionsDistinct$

White list of reasons

There exist two possible classes of allowed reasons for breaking the constraints. If the specific reason can be argued to be part of one of the higher class reasons it is acceptable that the constrain is broken.
Non existing properties: It can be the case that the constraint breaks since files are missing that would be required for the constrain to hold. Hence, it is in fact a correct behavior of the tool that the constraints does not hold. For the project itself it means that it may contains a bug.
Out of scope: The coded tool made some assumptions to reduce the possible cases to deal with. Therefore, our research defines a scope. That scope for example does not include some *Spring* related packages for the execution of the O/R-Mapping. If the tool does not detect a property because the file was out of scope it should be acceptable. To cover the complete scope would need much more effort and hence it is restricted in the case study.

Discussion of the result

The collected metrics of the tool that are related to the constraints are illustrated in Table 10. A few constraints were broken in the projects but for all of those breaks a reason corresponding to the explanations given in the whitelist chapter (5.4.2) exists. Therefore, it can be said that the evaluation with this approach has been successful. A detailed list of the analysis for each project is available on GitHub [9].

Well defined function constraint broken:
All of the not well formed mapping functions have missing *RLangs*. The following paragraph explains the reasons for the absence of the language in the projects.

Partly missing SQL statements: Projects like *Oscar* and *Eeg Database* are defining several *RLangs* but not as much as *OLangs*. The reason for this lies in two facts. On the one hand, fifteen SQL files are missing so that in fact those tables are never defined. On the other hand, some files are using not supported SQL elements which would be out of scope to the parser, e.g., the *ENABLE* statement. This was

[9] https://github.com/fruether/RuleEngine/blob/master/Dataset/AnalysisOfMissingFunctions.xlsx

counted five times. Hence it is a combination of *out of scope* and *missing files* if we consider the defined white list.

Hibernate based project setup: There exists a feature that empowers Hibernate to automatically create the database schema. This option can be enabled by setting the property *hibernate.hbm2ddl.auto* to *update* or *true* in either the *persistence.xml* file or another Hibernate related configuration file. This was done for *frontlinems-core* and *Oltpbench* and as a result the missing *RLangs* are white listed according to *missing files*.

Own technology: The project *projectforge-webapp* is using an own Java based framework for the creation of the SQL tables, which is called *projectforge-continuous-db*. Therefore, it seems to be reasonable that no SQL files were found in the project. This counts as white listed according to *out of scope*.

Unknown database creation: *Druid, IBPMiddleware, Mateo and iTest* do not provide any hint on how the database schema is created. There exists no single SQL file (with a create statement for the requested tables) in the repository and hence our approach is not broken. However, Mateo is even setting in *mateo.properties* the property *hibernate.hbm2ddl.auto* to none. For the other projects no proof was found that it was set to *update*. Neither IBPMiddleware nor Mateo could pass the tests while building them with maven. This may indicates a hidden bug in the system. The broken constraints count as white listed according to *missing files*.

Function usage constraint broken:

This constraint is broken because the function usage is below the amount of defined mappings. The following reasons were found in the projects for breaking the constraint.

Spring framework:

The Spring framework manifests different ways to save Java entities to the database, e.g., by using *org.springframework.jdbc.core.JdbcTemplate*. Since our approach focuses on core Hibernate, it was not possible to detect all mapping operations defined by other libraries. This is the case for the projects *projectforge-webapp* and *Druid* and counts up to three missing detections of function usage. It can be classified as *out of scope*.

Missing files:

Some function usages were simply never implemented. Hence it is logical that they are not found by our tool. This is the reason for broken constraints in *Oscar*, *ROMS*, *projectforge-webapp*, *Mateo*, *Oltpbench*, *frontlinesms-core* and *iTest*. All in all, this is the cause of sixteen breaks. It can be assigned to the white list element named *not existing*.

Missing type system

There exists one case where a service is returning a List[10] without specifying the concrete elements stored in it. As a result, it is not possible without a type system to detect the actual type. For this case our approach fails to detect the real value. This was taking place for *Oscar* and it is *not in our scope*.

[10] https://docs.oracle.com/javase/8/docs/api/java/util/List.html

Correspondence constraint broken:
The reasons this is broken is caused by the missing *RLangs* which were explained in the previous section.

Project	Hibernate Mappings	OLangs	RLangs	Corres pondence	Implemented functions
Oscar	484	484	475	1687	478
ROMS	35	35	35	36	34
Druid	2	2	-	-	1
projectforge-webapp	67	67	-	-	61
IBPMiddleware	71	71	-	-	71
EEG Database	59	59	47	64	59
Mateo	100	100	1	1	97
Oltpbench	7	7	-	-	3
frontlinesms-core	13	13	-	-	12
iTest	1	1	-	-	-

Table 10 Summary of the result for the internal validation

5.4.3 External Validation

The result of the tool should reflect the state of the evaluated project. In this chapter, certain properties are checked against another data set based on the summary provided by the metrics. In case the project has a different state than the tool calculated, e.g., having more or less mapping definitions than found by the tool, the rules seem to contain a bug. This way it can be validated that no important file is missing which is not checked by the internal approach explained in chapter 5.4.2. Otherwise, it does not look at the relations and semantics between the extracted metrics itself. Hence both approaches complement each other.
The approach calculates the state of the project with the help of a data set provided by an external source, a coded python script. Since it is validated against a different source it is called external validation.

Properties that should be checked

The mega model depends on the correctness of several relations. The most important constructs are the mapping definitions because all the further extractions are based on them. Without the identification of a mapping file no function would be identified and no function usage could be observed. Therefore, all the mapping files should be found by the tool. The second important property is itself related to the function usage. Each use of a O/R-mapping function should be detected and it is assumed that a valid Hibernate DAO service should implement at least one mapping function. The following list of the questions is derived from this discussion and should be checked in this approach:

1. Are all the object mapping definitions found?
2. Are all the *Hibernate DAO* services identified (true negative)?
3. Are the identified function usages really taking place (false positive)?

Search base validation

The most optimistic and naive approach for the finding of a *HibernateMapping* definition is to simply search for the files with the suffix *.hbm.xml*, which is the expected ending of a XML based Hibernate mapping file[11]. Another possibility to identify mapping files is to search in *Java* files for the content *@Entity*[12], which is the annotation the Java file should contain. For the identification of the files that use one of the Hibernate mapping functions, best practices are taken into consideration. Those practices assume that the Java class, that is executing a mapping, ends with the suffix *DAO* or *Dao*.
This simple search base approach is of course not perfect since the content could also exist in different scenarios that have no connection to Hibernate. However, the result provided by this naive approach is a good oracle for the data that have been extracted. In fact, if a file is found by the oracle approach but not by the tool an explanation for the missing has to be provided. The files that were just found by the tool should as well be examined. If the files are matched by both approaches confidence can be gained that they were correctly classified and they are not further investigated.
The naive approach collects its data with the help of Python scripts that either scan the file name or the content of a file.

Result

Mapping identification: The result of the performed naive approach for the identification of *HibernateMapping* file can be seen in Table 11. It shows that both approaches found exactly the same amount of mapping descriptions in the project. This supports the correctness of our tool since there is no evidence that a file is missing.

[11] https://docs.jboss.org/hibernate/orm/3.6/quickstart/en-US/html/hibernate-gsg-tutorial-basic.html
[12] https://docs.oracle.com/javaee/6/api/javax/persistence/Entity.html

Project	Oracle XML	Oracle Java	Tool XML	Tool Java
Oscar	47	437	47	437
ROMS	35	0	35	0
Druid	0	2	0	2
projectforge-webapp	0	67	0	67
IBPMiddleware	0	71	0	71
EEG Database	0	59	0	59
Mateo	0	100	0	100
Oltpbench	0	7	0	7
frontlinesms-core	0	13	0	13
iTest	0	1	0	1

Table 11 Comparison of the results between the tool and naive approach

Function usage identification (true negative and false negative):
The performing of the native approach on the repository found several files that are according to their name a *DAO*. The metric *FunctionsUsingFile*, which contains files that are using a mapping function, was used to get the result of the tool. Table 12 is showing a comparison of the files found by both approaches. The files that were found by the naive approach but not by the tool were examined and reasons were found for the missing of the files. Some were in fact correctly not classified, others were not found because of the limitations of our approach. In the end, the deviation could be explained and more confidence regarding the behaviour of the approach was gained. The analysis of the reasons for the not classification was performed and is documented in the next paragraph.

Result:

- The file is defining an interface or the class itself is empty (150 files and is a true negative case)
- The file is using another DAO and it not directly using the mapping functions of Hibernate (22 files and true negative case)
- For some other reason there is no mapping performed (three times and true negative)
- The way the files is using the objects is impossible to detect without a type system, e.g., they just return an object or a list without a template (seven times and it a false negative)
- A Spring related package is imported by the file which is not supported by the approach (six times and a false negative case)

Since the *false negatives* are the minority cases (7 %) and all are out of scope for this version, it can be argued that there exist no heavy problems for this metric. However, the found issues have to be tackled in future work.

Project	FunctionsUsingFile	N: DAOs	N: DAOs (no interfaces)
Oscar	471	500	496
ROMS	19	40	20
Druid	1	6	2
projectforge-webapp	103	77	76
IBPMiddleware	81	74	74
EEG Database	48	80	45
Mateo	180	88	15
Oltpbench	1	0	0
frontlinesms-core	10	24	13
iTest	0	0	0
Summary	904	889	741

Table 12 The value of the metric and the result of the search of the DAO files with the naive (N) approach

Function usage identification (false positive and true positive):
The other property that should be checked was to look for the false positive cases in the identified function usages. Since it is to much work to check each of the retrieved files manually, the manual check was restricted to the files that were only classified by the tool but not by the naive (search base) approach. This reduced the amount of checked files and the result of it can be seen in Table 13. It is notable that overall a lot wrong classifications happened. However, this is mostly driven by the project *projectforge-webapp*, which included 80 per cent of false positives.

Project	Correctly classified (TP)	Wrongly classified (FP)
Oscar	1	0
ROMS	0	0
Druid	1	0
projectforge-webapp	0	42
IBPMiddleware	9	7
EEG Database	1	2
Mateo	1	0
Oltpbench	1	0
frontlinesms-core	0	0
iTest	0	0
Summary	14	51

Table 13 Shows how many files, not covered by the naive approach, are correctly and wrongly classified by the tool

This project is special because all the test cases are extending a class called *AbstractTestBase*. This abstract class is implementing some Hibernate related operations, e.g., cleaning the database for the setup of a test case. Therefore, this file is using Hibernate mapping functions. The issue is that based on our definition all classes that inherit from *AbstractTestBase* are as well using the same packages. It happens to be that almost all tests cases inherit from it and as a result all files that in addition using a persistable object are classified as executing the mapping function. It is arguable if this is valid since a class would also inherit the operation of its super class. However, a call tree in combination with the type system would have to be implemented to solve this issue. This is for now out of scope but yet those classifications are not correct.
The other error cases are more acceptable since the files use a *DAO* which as well is implementing a method called *saveOrUpdate*. Furthermore, a session is declared. Hence the classification is wrong but not complete noise since the files are still closely related to the execution of a Hibernate mapping procedure.

5.4.4 Notable Examples

The limits of the extractions were illustrated by the evaluation. An example of a wrongly classified Java file can be found by looking at Listing 37 which is part of the IBPMiddleware[13] project. The fragment of the source code is showing a function which is defining a *Session* object. In addition, it is declaring an object that should be persisted, namely of the type *GermplasmList*. Furthermore, a Dao service is declared which implements and defines a method called *saveOrUpdate*. Although the file does not perform a direct Hibernate mapping, all the conditions of the rule are matched and it gets wrongly classified. Eventually, the Hibernate mapping routines are executed in the called service method. This false positive error could be fixed by implementing a (lightweight) type system which is for now out of scope.
The other case that can happen is that a file is not indicating the use of a Hibernate mapping function but yet it is using one. This false negative error is for example the case for the fragment shown in Listing 38 that is extracted of a file in the Oscar[14] project. This file is not declaring any persistent object since it is just returning an array of *Objects*. The type of those objects is impossible to detect without using type system which is out of scope. Therefore, it was not possible to detect that this file is in fact implementing a certain Hibernate mapping function.

[13] https://github.com/IBPlatform/IBPMiddleware/blob/master/src/main/java/org/generationcp/middleware/manager/GermplasmListManagerImpl.java
[14] https://github.com/scoophealth/oscar/blob/master/src/main/java/org/oscarehr/common/dao/forms/FormsDao.javaL53

Listing 37 This fragment is an example for a false positive error

```
1  private List<Integer> addOrUpdateGermplasmList(List<GermplasmList> germplasmLists,
2                                                 Operation operation)
3                                                 throws MiddlewareQueryException {
4          Session sessionForLocal = getCurrentSessionForLocal();
5          // ......
6          GermplasmListDAO dao = new GermplasmListDAO();
7          // ......
8          dao.saveOrUpdate(germplasmList);
9          //....
10 }
```

Listing 38 This fragment is an example for a false negative case

```
1  @NativeSql("formLabReq07")
2      public List<Object[]> findIdFormCreatedAndPatientNameFromFormLabReq07() {
3          String sql = "SELECT ID, formCreated, patientName FROM formLabReq07";
4          Query query = entityManager.createNativeQuery(sql);
5          return query.getResultList();
6      }
```

5.4.5 Limitations

One of the limitation of the evaluation is that not all the metrics can be checked easily. For example the metric *ImplementedFunctions* could neither be validated by the internal nor by the external validation. This is the case for different reasons for both approaches. On the one hand, the number of function usages is to large to be manually checked. On the other hand, the usage itself is difficult to identify by an oracle. A simple search for content would be to broad to detect which function is used. Therefore, this property could not be validated by this approach.

5.5 Discussion

5.5.1 Usage of XML and Annotations

The mega model as well as the metrics show that rather Java annotations are used than XML files for the description of a mapping between an object and a table. Only the two projects *Oscar* and *ROMS* out of the ten examined choose to describe some mappings with the help of XML. Of those descriptions only *ROMS* exclusively relays on XML. For *Oscar* the majority of the mappings is described by annotations. As a result, it can be concluded that Hibernate is mostly used with the help of annotations. XML mapping does not play a large role in the sampled projects that are using Hibernate.

Project	Table statements	Correspondence	Mappings
Oscar	2137	1687	484
ROMS	106	36	35
Druid	39	0	2
Projectforge-webapp	0	0	67
IBPMiddleware	0	0	71
Eeg Databse	103	64	59
Mateo	10	1	100
Oltpbench	499	0	7
Frontlinesms-core	0	0	13
iTest	0	0	1

Table 14 Table statement, correspondences and mapping definitions of the projects

5.5.2 Database Creation in the Hibernate Context

Hibernate is a powerful tool for the mapping of objects to tables. However, there are several possibilities how the database schema with the required tables can be created. Each approach has its advantages and disadvantages. Those are heavily discussed on Stackoverflow, e.g., if Hibernate's automatic schema creation is safe in production [15]. The approach shows, that six projects actually contain *create table*[16] statements. However, the existing *TableStatements* have quite often a larger value than the metric counting the *MappingDefinitions*. This shows that there exist more tables in the projects than required by the mapped objects. A conclusion could be that there are either unused tables or some data are stored in another way than by executing O/R mappings.
Another notable property is that quite a few projects are using different non SQL based approaches for the setup of the persisted objects in the database schema. This is indicated by the fact, that not the same number of mappings and correspondence relations exist. The tool of us could yet indicate which SQL files may be involved in the initial process. In addition, it points out projects that use a different approach for the schema creation. An in-depth analysis of the used approaches for the creation of the schema can be found in the Evaluation chapter. Table 14 illustrates the usage of table statements, found correspondences and mapping definitions in the different projects.
To summarize, it can be said that only a few projects use SQL files for the initialization of the database and often there seem to be more tables than mapped objects.

5.5.3 Function Usage Identifications

The extracted mega model gives a hint about functions which have not been implemented. Sixteen missing function executions could be identified by our approach and were validated. In addition, the approach gave a good indication of the files that should be checked by a developer, if he is looking for the execution of a certain mapping function. The false positive rate is with sixteen elements better compared to the naive ap-

[15] https://stackoverflow.com/questions/221379/hibernate-hbm2ddl-auto-update-in-production
[16] https://www.w3schools.com/sql/sql$_c reate_t able.asp$

proach. Therefore, our result gives a satisfying set of files that should be examined, to find the execution of a specific mapping function.
However, the evaluation shows that the heuristic is far away from being perfect. There exist some cases that could not be detected by the approach at all. This issues have to be solved in future work.

6 Concluding Remarks

6.1 Summary

This work provides a rule based approach for the automatic extraction of a linguistic architecture with the help of *RDF triples* that are encoding the *MegaL* ontology. The open source framework Apache Jena is responsible for providing and executing the rule infrastructure. The complete approach is evaluated and tested with a case study.
In the case study a mega model using the MegaL ontology was created for the Hibernate framework. The illustration of this model can be seen in the chapter 5.2 , which lays out the required result of the extraction process. Afterwards, a set of rules was developed that should, with the help of built-ins, extract the relations of the mega model. Those rules were explained in depth and afterwards executed on sample projects. Those sample projects were selected with the help of a well defined methodology, which is based on previous work. Finally, the extracted models are summarized with the help of metrics which are then evaluated in depth with the help of an oracle and manual verification. The case study shows that it is possible to extract mega models for a Hibernate scenario with our rule based approach. In addition, knowledge about a projects structure and it special cases can be gained.

6.2 Threat to Validity

6.2.1 Internal Validity

Internal validity is related to the execution of the experiment. To avoid this kind of a threat all the samples were executed with the same version of the tool. That way a change of the result based on a change of the source code is not possible. On top of this, all project files were included as input. Hence only the input changed but nothing else. The project files itself were downloaded from the GitHub master branch of the projects repository. Therefore, there should be no confound variable during the execution. The same project files were used for the naive approach.
One threat is that not all the metrics could be validated by the oracle. For example the metric *ImplementedFunctions* could not be verified since there exists no easy approach to check this condition. The oracle would have to parse the file itself which would itself be prone to errors. Therefore, this condition could not be checked and hence there is maybe a discrepancy between the calculated and real result for this metric.

6.2.2 External Validity

External validity is concerned with the ability to generalize the results of the experiment. The projects were selected from GitHub and hence all of those count as real world projects. They are not small regarding contained classes and are developed in a team with the help of the Hibernate technology. Therefore, those projects should be a good selection to generalize on. However, it could be the case that due to unknown reasons certain features of Hibernate were not used in any of those projects, e.g., a certain mapping function. Therefore, a threat to the the external validity is that certain features, e.g., a special create statement, could not be tested in this sample. As a result, the tool may breaks on projects where those features are present.

6.2.3 Construction Validity

Construction validity is related to the design of the experiment. One of the goals is to show that a mega model can automatically be extracted and interesting properties related to technology usage can be shown. Since the case study design is only considered with Java projects and with the Hibernate framework, it can be argued that this is not enough to conclude that a mega model can be extracted in different scenarios. After all Hibernate and Java could be so specific in their characteristics that only this combination enables the extraction of a mega model. Therefore, a countermeasure would be to run such an experiment in another setup that is independent of Hibernate, e.g., that extracts a Spring related mega model.

6.2.4 Conclusion validity

Conclusion validity is related to the analysis. The analysis is performed by the metrics and the tool. The coded tool is implemented in Java. It could happen that errors appear in it which may lead to a wrong output or a wrong summary of the metrics. A countermeasure for this validity threat was developed by implementing test cases that check the result of the extensions and the services. Furthermore, the metrics were tested. As a result, certainty can be gained that the analysis of the tool was correct and reflects the intended behavior.

6.3 Future Work

Precision improvement for function usage:
There are several measurements that can be performed to achieve a better false positive rate. The most promising approach is to implement a type system that checks the type of the object which is either the parameter or the return value of an O/R mapping execution call. In addition, it would be checked if it is really the method in question or if it just happens to have the same name. This way it should be possible to eliminate a bunch of the classification errors. In addition, a call tree could help to make the inheritance of package usage a bit more clear and detailed. Last but not least, the *MegaL* model can be made more precise by splitting the Java files into different fragments

that define a method. That way it could be investigated which method is calling which mapping function. As a result, the developer can get a more in depth summary and eventually more insights regarding the projects architecture.

Correspondence based improvements:

Another field of future work would be to add more SQL dialects and database management systems to the approach. Alternatively the correspondence between tables and classes could be sophisticated by showing the correspondence on the column and attribute level. This would lead to an identification of deprecated SQL statements, that do not create required columns. Since not all projects use SQL files for the creation of the tables representing the objects the earlier described approach would be of more value.

Evaluation improvement:

Right now not all the metrics could be validated. For example, there is no approach to check how high the precision of *ImplementedFunctions* is since there is no easy way to validate if a file is implementing a Hibernate function. Therefore, in future work a strategy for the validation of this metric has to be developed and applied. A possible approach would be to count how many functions a certain file is implementing and then validate the files that implement a high amount of different functions. If they wrongly assign a function usage to a file the reason has to be systematically analyzed and documented. This could lead to the discovery of patterns and in the end lead to a strategy to reduce the miss classifications .

7 Appendix

7.1 SQL commands

Listing 39 Getting final hibernate versions

```
Select
    count(p_D.id) as elements,
    a_v.version as longerversion
from
    api_version as a_v, Project_Dependency as p_D
where
        a_v.version like '%Final'
        and dp_D.version = a_v.version
group by(a_v.version) order by elements;
```

Listing 40 Getting projects with fitting versions and classes

```
Select
    project_name as longername,
    count(class_name) as classes
from
    Project_Dependency as p_D1,
    Projects as p1, Classes as c
where
    p1.id = p_D1.id
    and c.pr_id = p1.id
    and p_D1.version in
    (Select
        p_D.versionfrom api_version as a_v,
        Project_Dependency as p_D
        where
            a_v.version like '%Final'
            and p_D.version = a_v.version
        group by(p_D.version)
        order by count(p_D.id) DESC limit 10)
group by (project_name)
having count(class_name) > 200
order by classes DESC;
```

Listing 41 SQL for linking projects with API usage

```
Select
    count(project_name) as longername
from
    Project_Dependency as p_D1,
    Projects as p1
where
    p1.id = p_D1.id
    and p_D1.version in
```

```
    (Select
        p_D.version
        from
            api_version as a_v,
            Project_Dependency as p_D
        where
            a_v.version like '%Final'
            and p_D.version = a_v.version
        group by(p_D.version)
order by count(p_D.id) DESC limit 10);
```

Listing 42 Multiple persons should work on a Hibernate project

```
Select
    project_name,
    count(class_name) as classes
from
    Project_Dependency as p_D1,
    Projects as p1, Classes as c
where
    p1.id = p_D1.id
    and c.pr_id = p1.id
    and p_D1.version in
    (Select
        p_D.versionfrom api_version as a_v,
        Project_Dependency as p_D
    where
        a_v.version like '%Final'
        and p_D.version = a_v.version
    group by(p_D.version)
    order by count(p_D.id) DESC limit 10)
    and c.pr_id in
        (Select DISTINCT
            pr_id
        from
            Class_History as ch,
            Classes as c1
        where ch.cl_id = c1.id
        group by pr_id
        having 10 < count(DISTINCT ch.author_name))
group by (project_name) having count(class_name) > 100
order by classes DESC
```

7.2 Further MegaL Definitions

7.2.1 Further Entities

Language	Description	Origin	Hyperlink
SQLCreate-TableStmt	A language that is defining the SQL create table statement	SQL standard	Wikipedia
JavaClassURI	A class is a common concept in object oriented programming. In the context of Java a class is identified at runtime by its name and its package. This is the language the identifier is following.	Java specification	Wikipedia
JavaAnnotated-Element	The language that describes a Java annotation and the element it is connected to.	Java specification	Wikipedia
QualifiedName	An unambiguous name that identifies an element of the system. It is a special reference language.	MegaL	Wikipedia
Artifact	**Description**	**Origin**	**Hyperlink**
Package	Organizes Java packages. In the end, it is usually a directory.	Java specification	Wikipedia

Table 16 Entities depending on the ecosystem used by software engineering

7.2.2 Newly Added MegaL Relations

defOccurs	Artifact # Artifact	The content of a entity can declare a reference name to identify this artifact. For example a Java class file declares a class identifier.	New in MegaL
refOccurs	Artifact # Artifact	The content of a entity can reference another artifact. Usually the identifier should be declared by a *refOccurs* relation.	New in MegaL
input	Language # Function	A function is taking input in the form of a certain language.	Used only for this approach
output	Function # Language	A function returns a certain language if a certain input was given	Used only for this approach

7.2.3 Sub-properties in RDF

Name	**Parent**	**Description**	**Origin**
HibernateMapping RefOccuresRelationalTable	refOccurs	Makes clear that a Hibernate mapping file refers to relational table.	New

7.3 API description of Hibernate

7.3.1 Required packages

1. org.hibernate.cfg.Configuration
2. org.hibernate.Session
3. org.hibernate.SessionFactory
4. javax.persistence.Query
5. org.springframework.orm.hibernate3.support.HibernateDaoSupport
6. javax.persistence.EntityManager
7. org.hibernate.Criteria
8. org.hibernate.Query

7.3.2 Required function calls for the mapping execution

1. createSQLQuery of *org.hibernate.Session*
2. iterate of *org.hibernate.Query*
3. createCriteria of *org.hibernate.Session*
4. list of *org.hibernate.Query*
5. executeUpdate of *org.hibernate.Query*
6. getFirstResult of *org.hibernate.Criteria*
7. getResultList of *org.hibernate.Criteria*
8. getSingleResult of *org.hibernate.Criteria*
9. unwrap
10. delete of *org.hibernate.Session*
11. findByNamedParam of *org.springframework.orm.hibernate.HibernateTemplate*
12. get of *org.hibernate.Session*
13. save of *org.hibernate.Session*
14. persist of *org.hibernate.Session*
15. refresh of *org.hibernate.Session*
16. replicate of *org.hibernate.Session*
17. saveOrUpdate of *org.hibernate.Session*
18. update of *org.hibernate.Session*
19. uniqueResult of *org.hibernate.Criteria*

References

[1] Sultan S. *Al-Qahtani*, Ellis E. *Eghan*, and Juergen *Rilling*. "Recovering Semantic Traceability Links between APIs and Security Vulnerabilities: An Ontological Modeling Approach". In: *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*. 2017, pp. 80–91. DOI: 10.1109/ICST.2017.15. URL: https://doi.org/10.1109/ICST.2017.15.

[2] Nicolas *Anquetil*, Káthia Marçal de *Oliveira*, Kleiber D. de *Sousa*, and Márcio Greyck Batista *Dias*. "Software maintenance seen as a knowledge management issue". In: *Information & Software Technology* 49.5 (2007), pp. 515–529. DOI: 10.1016/j.infsof.2006.07.007. URL: https://doi.org/10.1016/j.infsof.2006.07.007.

[3] J. *Arthur* and S. *Azadegan*. "Spring framework for rapid open source J2EE Web application development: a case study". In: *Sixth International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing and First ACIS International Workshop on Self-Assembling Wireless Network*. 2005, pp. 90–95. DOI: 10.1109/SNPD-SAWN.2005.74.

[4] Jean-Marie *Favre*, Ralf *Lammel*, Martin *Leinberger*, Thomas *Schmorleiz*, and Andrei *Varanovich*. "Linking Documentation and Source Code in a Software Chrestomathy". In: *WCRE*. IEEE Computer Society, 2012, pp. 335–344. ISBN: 978-1-4673-4536-1.

[5] Jean-Marie *Favre*, Ralf *Lämmel*, and Andrei *Varanovich*. "Modeling the Linguistic Architecture of Software Products". In: *Model Driven Engineering Languages and Systems: 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30–October 5, 2012. Proceedings*. Ed. by Robert B. *France*, Jürgen *Kazmeier*, Ruth *Breu*, and Colin *Atkinson*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 151–167. ISBN: 978-3-642-33666-9. DOI: 10.1007/978-3-642-33666-9_11. URL: https://doi.org/10.1007/978-3-642-33666-9_11.

[6] Charles L. *Forgy*. "Expert Systems". In: ed. by Peter G. *Raeth*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1990. Chap. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem, pp. 324–341. ISBN: 0-8186-8904-8. URL: http://dl.acm.org/citation.cfm?id=115710.115736.

[7] Hans-Jörg *Happel* and Stefan *Seedorf*. "Applications of Ontologies in Software Engineering". In: *International Workshop on Semantic Web Enabled*

Software Engineering (SWESE'06). Athens, USA, 2006. URL: http://fparreiras/papers/AppOntoSE.pdf.

[8] Johannes *Härtel*, Lukas *Härtel*, Ralf *Lämmel*, Andrei *Varanovich*, and Marcel *Heinz*. "Interconnected Linguistic Architecture". In: *CoRR* abs/1701.08122 (2017). URL: http://arxiv.org/abs/1701.08122.

[9] Ralf *Lämmel* and Andrei *Varanovich*. "Interpretation of Linguistic Architecture". In: *Modelling Foundations and Applications: 10th European Conference, ECMFA 2014, Held as Part of STAF 2014, York, UK, July 21-25, 2014. Proceedings*. Ed. by Jordi *Cabot* and Julia *Rubin*. Cham: Springer International Publishing, 2014, pp. 67–82. ISBN: 978-3-319-09195-2. DOI: 10.1007/978-3-319-09195-2_5. URL: https://doi.org/10.1007/978-3-319-09195-2_5.

[10] W. *Meng*, J. *Rilling*, Y. *Zhang*, R. *Witte*, and P. *Charl*. "An Ontological Software Comprehension Process Model". In: *In Proc. of the 3rd International Workshop on Metamodels, Schemas, Grammars, and Ontologies for Reverse Engineering*. 2006.

[11] C. *Nagy*, L. *Meurice*, and A. *Cleve*. "Where was this SQL query executed? a static concept location approach". In: *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 2015, pp. 580–584. DOI: 10.1109/SANER.2015.7081881.

[12] D. *Ratiu*, M. *Feilkas*, and J. *Jürjens*. "Extracting Domain Ontologies from Domain Specific APIs". In: *12th European Conference on Software Maintenance and Reengineering (CSMR 08)*. IEEE. 2008, pp. 203–212. DOI: http://dx.doi.org/10.1109/CSMR.2008.4493315.

[13] Coen De *Roover*, Ralf *Lämmel*, and Ekaterina *Pek*. "Multi-dimensional exploration of API usage". In: *IEEE 21st International Conference on Program Comprehension, ICPC 2013, San Francisco, CA, USA, 20-21 May, 2013*. 2013, pp. 152–161. DOI: 10.1109/ICPC.2013.6613843. URL: https://doi.org/10.1109/ICPC.2013.6613843.

[14] Anand Ashok *Sawant* and Alberto *Bacchelli*. "A Dataset for API Usage". In: *Proceedings of the 12th Working Conference on Mining Software Repositories*. MSR '15. Florence, Italy: IEEE Press, 2015, pp. 506–509. ISBN: 978-0-7695-5594-2. URL: http://dl.acm.org/citation.cfm?id=2820518.2820599.

[15] M. *Uschold* and M. *Gruninger*. "Ontologies: principles, methods and applications". In: *The Knowledge Engineering Review* 11.2 (1996), pp. 93–136.

[16] René *Witte*, Yonggang *Zhang*, and Juergen *Rilling*. "Empowering Software Maintainers with Semantic Web Technologies". In: *The Semantic Web: Research and Applications, 4th European Semantic Web Conference, ESWC 2007, Innsbruck, Austria, June 3-7, 2007, Proceedings*. 2007, pp. 37–52. DOI: 10.1007/978-3-540-72667-8_5. URL: https://doi.org/10.1007/978-3-540-72667-8_5.

[17] Yonggang *Zhang*, René *Witte*, Juergen *Rilling*, and Volker *Haarslev*. "Ontological approach for the semantic recovery of traceability links between software artefacts". In: *IET Software* 2.3 (2008), pp. 185–203.