UNIVERSITÄT
KOBLENZ · LANDAU

Trace Link Recovery using Static Program Analysis

Bachelorarbeit

zur Erlangung des Grades eines Bachelor of Science
im Studiengang Informatik

vorgelegt von

Maximilian Meffert

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

Koblenz, im Dezember 2017

Erklärung

Hiermit bestätige ich, dass die vorliegende Arbeit von mir selbständig verfasst wurde und ich keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet–Quellen – benutzt habe und die Arbeit von mir vorher nicht in einem anderen Prüfungsverfahren eingereicht wurde. Die eingereichte schriftliche Fassung entspricht der auf dem elektronischen Speichermedium (CD-Rom).

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

....

(Ort, Datum) (Maximilian Meffert)

Zusammenfassung

Moderne Software Systeme sind heterogen bezüglich der verwendeten Sprachen und Technologien, welche zur Programmierung und Inter-Prozess-Kommunikation verwendet werden, was eine Herausforderung für das kognitive Verständniss von Programmen darstellt. Untersützung von Traceability (Rückverfolgbarkeit) kann helfen diese Herausforderung zu meistern, indem verwandte Komponenten eines Software Systems hervorgehoben werden. Diese Abschlussarbeit skizziert einen Ansatz zur Wiederherstllung von Traceability zwischen Quellcodeartefakten mittles statischer Programmanalyse. Hierfür wird ein exemplarisches System zur Wiederhrestllung von Trace Links (Spurverküpfungen), welches Trace Links mit der Semantik von Linguistic Architectures (linguistische Architekturen) wiederherstellt, entwickelt und mit einer Mini-Fallstudie evaluiert.

Abstract

Modern software system are heterogeneous in terms of languages and technologies used for programming and inter process communication, which is a challenge for program comprehension. Support of traceability can provide help to overcome this challenge by bringing out related components of a software system. This thesis outlines an approach for recovering traceability among source code artifacts by utilizing static program analysiss. For this, an exemplary system for recovering trace links with the semantics of linguistic architectures is developed and evaluated with a mini case-study.

Acknowledgements

I want to express my sincere gratitude to my family, my mother, father and sister, for all the support they gave me throughout my education.

I also want to sincerely thank my supervisors. Thanks to Prof. Dr. Ralf Lämmel for providing inspirational new insights into the world of software. Thanks to Johannes Härtel for providing constructive feedback throughout the development of this thesis.

Last but not least, I want to sincerely thank my boss, Timo Ziegler, for providing conditions allowing me to work and study in parallel - and for a gentle push to get things done.

Contents

List of Theorems

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Modern software system are heterogeneous in terms of languages and technologies used for programming and inter process communication, which is a challenge for program comprehension. Support of traceability can provide help to overcome this challenge by bringing out related components of a software system.

1.1 Motivational Example

Common tasks in software development are implementation of serialization and persistence of domain models. For instance, consider a simple web service which serves data via HTTP (Hypertext Transfer Protocol) as XML (Extensible Markup Language) and stores it in a relational database. We call this an O/R/X-Mapping (Object-Relational- and XML-Mapping) scenario. Given such a system, the same conceptual data, i.e. the domain model, is transported through application tiers in different forms, that is, the same data is represented by various manifestations at a time. Each manifestation involves another software language and technology. Figure 1.1 opposes model- to instance-level syntaxes of different manifestations for a conceptual model of employees in an O/R/X-Mapping scenario. Employee only have a *name* and an *age* attribute. Persistence is presented in tuple notation, data transfer objects of an application tier are represented with a fictional high level programming language object notation and serialization is exemplified with XML and XSD (XML Schema Definition) syntax.

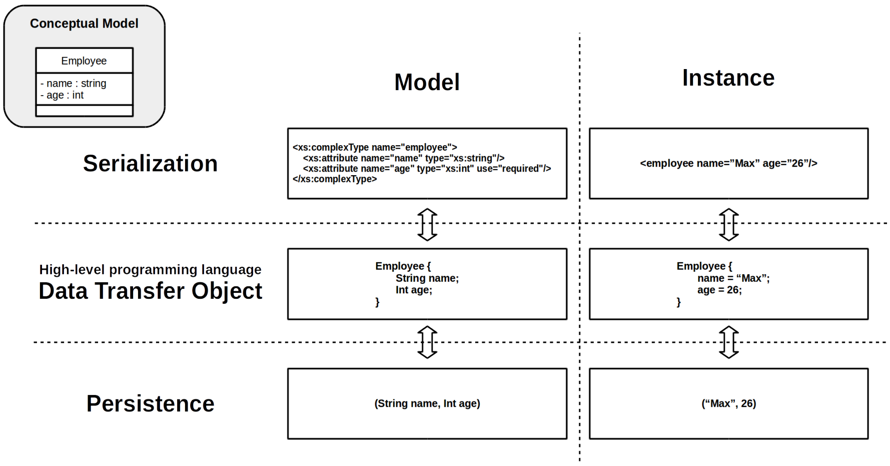

Conceptual Model
Employee
- name : string
- age : int
Model
Instance
Serialization
<xs:complexType name="employee">
<xs:attribute name="name" type="xs:string"/>
<xs:attribute name="age" type="xs:int" use="required"/>
</xs:complexType>
<employee name="Max" age="26"/>
High-level programming language
Data Transfer Object
Employee { String name; Int age; }
Employee { name = "Max"; age = 26; }
Persistence
(String name, Int age)
("Max", 26)

Figure 1.1 O/R/X-Mapping Manifestations

Although figure 1.1 shows a fictionalized example, one can observe structural similarities among the different model and instance representation syntaxes. Such similarities also occur in real-world software engineering through use of conventions which may be predefined in technologies for O/R/X-Mapping. Figure 1.2 displays an annotated Java class and XML/XSD output generated by JAXB (Java Architecture for XML Binding). Except for the root element all names for

```
@XmlRootElement(name="employee")
@XmlAccessorType(XmlAccessType.FIELD)
public class Employee {

    @XmlAttribute
    private int id;

    @XmlAttribute
    private String name;

    @XmlAttribute
    private int age;

    @XmlAttribute
    private double salary;

    private Department department;

    private Department managedDepartment;

}
```

```
<xs:complexType name="employee">
 <xs:attribute name="id" type="xs:int"
       use="required"/>

 <xs:attribute name="name" type="xs:string"/>

 <xs:attribute name="age" type="xs:int"
       use="required"/>

 <xs:attribute name="salary" type="xs:double"
       use="required"/>

 <xs:sequence>

   <xs:element ref="department" minOccurs="0"/>

   <xs:element name="managedDepartment"
         type="department" minOccurs="0"/>

 </xs:sequence>
</xs:complexType>
```

```
<employee name="Max" age="26" salary="55000.0"/>
```

Figure 1.2 JAXB XML/XSD Mapping

attributes and elements are taken from the Java class. Moreover, the XSD complex type and the Java class share a similar nested structure. Links through similarity can be found within the model-level representations (Java-XSD) and between model- and instance-level manifestations (Java-XML and XSD-XML), for instance:

- `public class Employee {...}` is linked with `<xs:complextType name="employee">...</xs:complexType>` and `<employee .../>`, the latter two are also linked with each other
- `private String name;` is linked with `<xs:attribute name="name" type="xs:string"/>` and `name="Max"`, the latter two are also linked with each other

1.2 Objectives

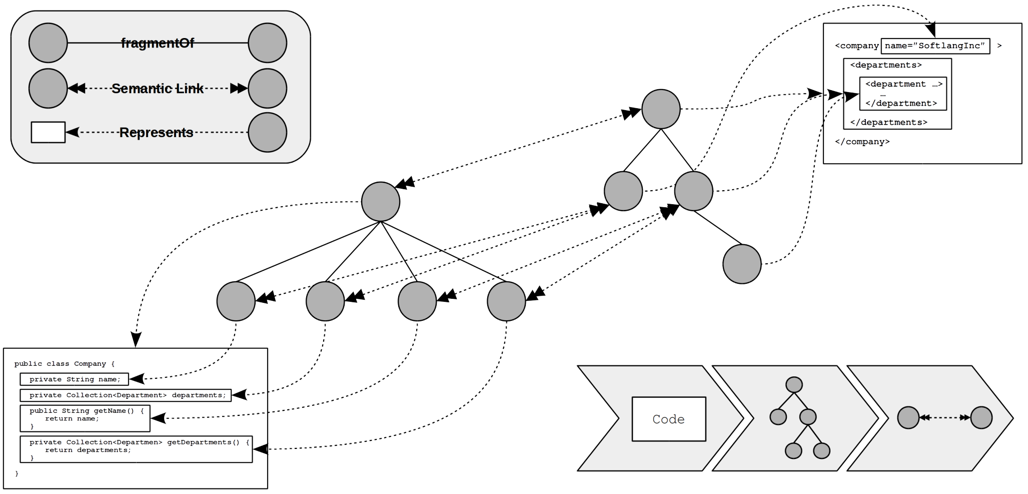
The aim of this thesis is to provide automated recovery of such similarities as semantic links. Recovered links are inserted into *megamodels* [4] for *linguistic architectures* [9] [20] [15]. Megamodels are models providing a high level of abstraction with other models as modeling elements, e.g. a megamodel may describe the dependencies between metamodels, models and instances. Linguistic architectures intend to describe software systems from a language centric point of view. They model knowledge about software systems in terms of languages, artifacts, technologies, etc. [14] [9] [20]. Such entities are interrelated with relationships derived from common software engineering and theoretical computer science vocabulary providing special semantics, e.g. *defines*, *isA*, *instanceOf*, *represents*, *implements*, *realizationOf*, *elementOf*, *subsetOf* [15].

Linguistic architectures are related to ER Models (Entity-Relationship Models) and ontologies. Recovering semantic links as described above is related to the concept of *traceability* and an application of *trace recovery* [13] (see §2.2). In context of traceability, semantic links may be called *trace links*, however, both establish a relation between two entities denoting a certain meaning.

Trace links among software artifacts denoting that one artifact encodes the same information as the other are called *correspondence* links and denoted *correspondsTo* (see §2.1.3). Trace links between two artifacts denoting that one defines the other, in the sense of a metamodel defining a model, are called *conformance* links and denoted *conformsTo* (see §2.1.4). The main objective is to recover such trace links among well-formed, possibly partial, source code artifacts. We call well-formed partial source code artifacts *fragments* (see §2.1.2).

1.3 Approach

Our approach for recovering trace links utilizes *static program analysis*. We use it in the broad sense, i.e. we construct specialized ASTs (Abstract Syntax Trees) for further analysis. Usually static program analysis is used with the purpose of formal verification or measurement of software engineering related properties of source code, e.g. cohesion and coupling metrics. We use static program analysis to semantically link code fragments and uncover knowledge of the analyzed artifacts. In that respect, our approach is related to computing software metrics.



```
<company name="SoftlangInc" >
  <departments>
    <department ...>
      ...
    </department>
  </departments>
</company>
```

```
public class Company {
  private String name;
  private Collection<Department> departments;
  public String getName() {
      return name;
  }
  private Collection<Departmen> getDepartments() {
      return departments;
  }
}
```

Figure 1.3 Recovery Approach

Figure 1.3 shows a schematic illustration of our recovery approach. It can be summarized with the following three steps:

1. we create two Parse Trees for both inputs respectively
2. Parse Trees are transformed to specialized ASTs; such ASTs are not intended to be used for compilation, instead each node represents a part of code which may be linked during further analysis; a child node represents

a piece of code which is properly embedded the code represented by its parent

3. we apply a comparative analysis to both ASTs; while traversing through both trees we check whether each pair of nodes is a trace link

1.4 Contributions & Non-Contributions

This section lists critical contributions and non-contributions of this thesis:

Contribution 1 We contribute a Java based API (Application Programming Interface) for recovering trace links among artifacts and their constituent fragments, providing utilities for static program analysis as described in §1.3 (see §4).

Contribution 2 We contribute an implementation for correspondence and conformance link recovery among JAXB and Hibernate artifacts utilizing the created API and axioms for linguistic architectures (see §5 and §2.1 respectively).

Contribution 3 We contribute a mini case-study applying the implemented recovery system to a minimal O/R/X-Mapping corpus (see §6).

Non-Contribution 1 We do not contribute to the axiomatization of linguistic architectures as described in [20], [19] and [15].

1.5 Road-Map

§2 provides the necessary theoretical and practical background for this thesis. §3 provides a summary on related work and positions this thesis among these works. §4 provides a high level design description for the system developed for this thesis. §5 provides a more detailed description of the implementation for crucial components of the developed system. §6 provides a mini case-study evaluating the developed system. §7 concludes the thesis.

Chapter 2

Background

This chapter provides the necessary theoretical an technological background topics of the thesis.

2.1 Axioms of Linguistic Architectures

This section summarizes axioms of linguistics architectures. Axioms are outlined in [19] and refined in [15]. §2.1.1 introduces the axiomatization for parthood §2.1.2

Axioms are presented in First Order Logic and formalization is taken from [15]. The universe to draw elements from is represented by the Entity-predicate:

$$\forall x.\mathsf{Entity}(x). \qquad \text{[15]}$$

We assume it holds for everything of interest, hence such things are called *entities*.

Linguistic architectures intend to describe software from language centric point of view, thus we provide specializations of entities for languages:

$$\begin{aligned} &\mathsf{Set}(x) \Rightarrow \mathsf{Entity}(x). && \text{[15]} \\ &\mathsf{Language}(x) \Rightarrow \mathsf{Set}(x). && \text{[15]} \end{aligned}$$

There are entities representing sets in a mathematical sense, and there are sets representing (formal-) languages in the sense of theoretical computer science.

On the other hand, we provide specializations for entities involved in software engineering terminology:

$$\text{Artifact}(x) \Rightarrow \text{Entity}(x). \qquad [15]$$
$$\text{File}(x) \Rightarrow \text{Artifact}(x). \qquad [15]$$
$$\text{Folder}(x) \Rightarrow \text{Artifact}(x). \qquad [15]$$

There are entities representing all kinds of digital artifacts, e.g. files and folders. Files represent persistent data resources, locatable either trough file systems or web services. Folders represent locatable collections of files. The intended use and semantic of these predicates is not meant to differ from intuitive, every day use.

2.1.1 Parthood

Parthood is the essential relationship when reasoning about correspondence and conformance among artifacts within linguistic architectures [19] [15]. It describes the relation between entities and their constituent parts. The study of parthood and its derivatives is mereology [30] [29]. In the context of linguistic architectures we assume most entities to be composed of several conceptual or "physical"[1] parts. In short, such entities exists as sum of their parts. That is, programs may compiled from many files, systems consists of several disjoint but dependent components, a Java class is made up of methods and fields, etc. Furthermore such entities are considered to be *mereologically invariant*, i.e. if one part changes, the whole changes as well. Axiom 1 captures parthood at its most basic level.

Axiom 1 (partOf)

$$\textit{partOf}(p, w) \Rightarrow \textit{Entity}(p) \wedge \textit{Entity}(w).$$
$$\textit{partOf}(p, w) \Leftarrow p \textit{ is a constituent part of } w.$$

The formalization presented here is not taken from [15], where it is not formalized explicitly, nor is it directly taken from [19]. It is rather a byproduct of merging works on linguistic architectures with works on mereology (see [19] [15] and [30]

[1] We mean physical in the sense, that an entity manifests in some (digital) form, but not in the sense that someone can touch it with bare hands

[29] respectively). Parthood is usually considered to be reflexive, antisymmetric and transitive, thus facilitating a partial order [30] [29]:

$$\mathsf{partOf}(p,p). \qquad \text{Reflxivity}$$

$$\mathsf{partOf}(p,w) \land \mathsf{partOf}(w,p) \Rightarrow p = w. \qquad \text{Antisymmertry}$$

$$\mathsf{partOf}(p,w) \land \mathsf{partOf}(w,u) \Rightarrow \mathsf{partOf}(p,u). \qquad \text{Transitivity}$$

Other authors consider partOf to be stricter, e.g. [19], tending towards an irreflexive parthood relationship. Irreflexive parthood is called proper in context of mereology [30] [29]:

$$\mathsf{properPartOf}(p,w) \Rightarrow \mathsf{Entity}(p) \land \mathsf{Entity}(w).$$

$$\mathsf{properPartOf}(x,y) \Leftarrow \mathsf{partOf}(x,y) \land \neg\mathsf{partOf}(y,x). \qquad \text{[30] [29]}$$

Proper parthood is the strict order induced by simple parthood [30] [29]. Figure 2.1 shows a schematic illustration opposing simple to proper parthood. This Venn-style diagram depicts two scenarios:

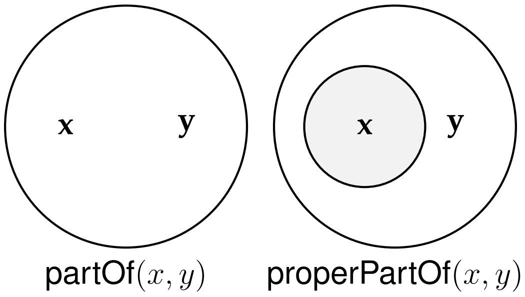


Figure 2.1 Simple vs. Proper Parthood

partOf(x,y) x may be a part of y or vice-versa; x and y cannot be distinguished

properPartOf(x,y) x is certainly a part of y, however y is not a part of x

In general proper parthood is a asymmetric relationship:

$$\mathsf{properPartOf}(x,y) \Rightarrow \neg\mathsf{properPartOf}(y,x). \qquad \text{[30] [29]} \qquad \text{Asymmetry}$$

In this thesis and the remaining axioms, we tend towards the terminology used by mereology for parthood relations, which provides a clear tool-set for distinguishing part-whole relationships. For instance, mereology also allows for the notion of atomicity, i.e. atomic parts which cannot be decomposed in further parts:

$$\mathsf{atomicPart}(x) \Leftarrow \nexists p.\mathsf{properPartOf}(p, x). \qquad \text{[30] [29]}$$

Atomicity is later used to distinguish cases for correspondence and conformance (see §2.1.3 and §2.1.4).

2.1.2 Fragments

Fragments are a specialization of entities intended to capture the endogenous, mereological decomposition of artifacts [15]. Axiom 2 defines fragments as artifacts, which are neither files nor folders, and are properly embedded in at least one other artifact. For instance, consider the syntactical decomposition of Java classes, i.e. the class fragment contains method and field fragments.

Axiom 2 (Fragment)

$$\begin{aligned}&\textit{Fragment}(f) \Rightarrow \textit{Artifact}(a) \wedge \neg(\textit{File}(f) \vee \textit{Folder}(f)).\\ &\textit{Fragment}(f) \Rightarrow \exists a.\textit{Artifact}(a) \wedge \textit{properPartOf}(f, a).\end{aligned}$$

We emphasize the proper parthood here for reasons mentioned in §2.1.1. If we were to use reflexive parthood for the definition of the Fragment-predicate it would be a tautology:

$$\mathsf{Fragment}(f) \Rightarrow \exists a.\mathsf{Artifact}(a) \wedge \mathsf{partOf}(f, a). \qquad \text{Tautology}$$

Since $\mathsf{Fragment}(f)$ implies $\mathsf{Artifact}(f)$ and partOf is reflexive, there exists always an artifact for a fragment the latter is a part of, which is the fragment itself.

From the specification of fragments we can specialize the proper parthood predicate for fragments:

$$\begin{aligned}&\mathsf{fragmentOf}(f, x) \Rightarrow \mathsf{Fragment}(f) \wedge \mathsf{Artifact}(x).\\ &\mathsf{fragmentOf}(f, x) \Leftarrow \mathsf{Fragment}(f) \wedge \mathsf{Artifact}(x) \wedge \mathsf{properPartOf}(f, x).\end{aligned}$$

This is just a restriction of domain and range facilitating a special semantic. A fragmentOf-relationship is not explicitly formalized in works on linguistic architectures, i.e. [19] [15].

We can also describe the process of fragmentation:

$$\mathsf{partOf}(p, w) \wedge \mathsf{Fragment}(w) \Rightarrow \mathsf{Fragment}(p). \qquad [15]$$

The fragment nature propagates top-down alongside the order of parthood, i.e. if an entity is a fragment all parts are also fragments [15].

2.1.3 Correspondence

Correspondence is the relationship between to artifacts denoting both represent the same data in the sense that they are mereologically similar [15]. It usually occurs as exogenous relationship, i.e. the involved artifacts do not belong to the same language, for instance XML and JSON (JavaScript Object Notation) may contain the same information.

In order to axiomatize correspondence properly we need to introduce two other relationships first, which are not explicitly defined in [15]:

represents The relationship denoting that an artifact is a representation or manifestation of an entity.

$$\mathsf{represents}(a, e) \Rightarrow \mathsf{Artifact}(a) \wedge \mathsf{Entity}(e).$$
$$\mathsf{represents}(a, e) \Leftarrow a \text{ is a representation of } e.$$

sameAs The relationship denoting two artifacts represent the same entity.

$$\mathsf{sameAs}(x, y) \Rightarrow \mathsf{Artifact}(x) \wedge \mathsf{Artifact}(y).$$
$$\mathsf{sameAs}(x, y) \Leftarrow \exists e.\mathsf{represents}(x, e) \wedge \mathsf{represents}(y, e).$$

The axiomatization of correspondence from [15] is slightly altered with emphasis of irreflexive proper parthood, see §2.1.1.

Axiom 3 (correspondsTo)

$$\begin{aligned}
&\textit{correspondsTo}(x,y) \Rightarrow \textit{Artifact}(x) \land \textit{Artifact}(y).\\
&\textit{correspondsTo}(x,y) \Leftarrow (\forall px.\textit{properPartOf}(px,x) \Rightarrow \exists py.\textit{properPartOf}(py,y) \land \textit{correspondsTo}(px,py))\\
&\qquad\qquad\qquad\qquad \land (\forall py.\textit{properPartOf}(py,y) \Rightarrow \exists px.\textit{properPartOf}(px,x) \land \textit{correspondsTo}(py,px))\\
&\qquad\qquad\qquad\qquad \lor (\nexists p.\textit{properPartOf}(p,x) \lor \textit{properPartOf}(p,y)) \land \textit{sameAs}(x,y).
\end{aligned}$$

Axiom 3 defines correspondence recursively:

Case I If both artifacts is atomic in the sense it cannot be decomposed in further parts, we check whether both artifacts represent the same data.

Case II Else, if at least one artifact is not atomic, then for each proper part of one artifact has to be a corresponding part in the other and vice versa.

This axiomatization demands a strict one-to-one correspondence in its recursive clause which occurs in reflexive cases, i.e. correspondsTo(x, x). A strict one-to-one correspondence may be to strict for real-world applications [19]. Correspondence inherits transitivity from represents and properPartOf.

2.1.4 Conformance

Conformance is the relationship between two artifacts denoting one satisfies the specification given by the other [15]. This satisfaction is also meant to be mereologically decomposable, i.e. there are parts of one artifact specified by a part of the other.

In order to axiomatize conformance properly we need to introduce two other relationships first. Both relations are not formalized in [15] as they are here:

defines The relationship simply denoting an artifact contains the definition of an entity.

$$\begin{aligned}
&\text{defines}(a,e) \Rightarrow \text{Artifact}(a) \land \text{Entity}(e).\\
&\text{defines}(a,e) \Leftarrow a \text{ is a definition for } e.
\end{aligned}$$

elementOf The relationship denoting set-theoretic membership.[2]

$$\mathsf{elementOf}(e,s) \Rightarrow \mathsf{Entity}(e) \wedge \mathsf{Set}(s).$$
$$\mathsf{elementOf}(e,s) \Leftarrow e \in s.$$

The axiomatization of conformance from [15] is also slightly altered with emphasis of irreflexive proper parthood; reasons for this are mentioned in §2.1.1.

Axiom 4 (conformsTo)

$$\textit{conformsTo}(a,d) \Rightarrow \textit{Artifact}(a) \wedge \textit{Artifact}(d).$$
$$\textit{conformsTo}(a,d) \Leftarrow (\forall pa.\textit{properPartOf}(pa,a) \wedge \exists pd.\textit{properPartOf}(pd,d) \wedge \textit{conformsTo}(pa,pd)) \vee \exists l.\textit{defines}(d,l) \wedge \textit{elementOf}(a,l).$$

Axiom 4 defines conformance recursively:

Case I If both artifacts is atomic in the sense it cannot be decomposed in further parts, we check whether one artifact defines a language the other artifact is an element of.

Case II Else, if at least one artifact is not atomic, then for each proper part of one artifact has to be a proper part in the other artifact it conforms to.

2.2 Traceability

This section provides brief background on the topic of traceability, i.e. its objectives and terminology (see §2.2.1 and §2.2.2 respectively).

2.2.1 Traceability Objectives

Traceability, as a desired property of software systems and their development processes, originates from requirement engineering, i.e. how do we validate or

[2] We do not use the elementOf axiom for artifacts and languages from [15], because it introduces the possibility for infinite mutual recursion:

$$\mathsf{elementOf}(a,l) \Rightarrow \mathsf{Artifact}(a) \wedge \mathsf{Language}(l).$$
$$\mathsf{elementOf}(a,l) \Leftarrow \exists s.\mathsf{defines}(s,l) \wedge \mathsf{conformsTo}(a,s)$$

verify that all requirements are met; or more precisely: on what data can we base such validation or verification? [31]

An answer to these questions is motivated by observations on development processes of software systems. Because such a process usually has an iterative and incremental nature, we can reasonably assume engineering activities to leave traces which reflect the performed action.

Modern day traceability objectives are not limited to validation of requirement compliance. It is also regarded as a desirable property for supporting engineering activities, e.g. change impact analysis or dependency analysis [13] [11]. This feeds to the overall topic of program comprehension. We can informally define traceability as *"[...] the ability to [...] interrelate uniquely identifiable entities in a way that matters."* [24] [31].

2.2.2 Traceability Terminology

This section recapitulates the necessary excerpt of definitions on traceability terms given by [13]. An excessive glossary of traceability terminology can be found in [12].

Definition 1 (Trace)

- ***(Noun)*** *A specified triplet of elements comprising: a source artifact, a target artifact and a trace link associating the two artifacts. [13]*
- ***(Verb)*** *The act of following a trace link from a source artifact to a target artifact or vice-versa. [13]*

Definition 2 (Trace Artifact, Source Artifact, Target Artifact) *A traceable unit of data (e.g., a single requirement, a cluster of requirements, a UML class, a UML class operation, a Java class or even a person). A trace artifact is one of the trace elements and is qualified as either a source artifact or as a target artifact when it participates in a trace:*

- ***Source Artifact*** *The artifact from which a trace originates.*
- ***Target Artifact*** *The artifact at the destination of a trace.*

[13]

Definition 3 (Trace Artifact Type) *A label that characterizes those trace artifacts that have the same or a similar structure (syntax) and/or purpose (semantics). For example, requirements, design and test cases may be distinct artifact types. [13]*

Definition 4 (Trace Link) *A specified association between a pair of artifacts, one comprising the source artifact and one comprising the target artifact. Trace links are effectively bidirectional. [13]*

Definition 5 (Trace Link Type) *A label that characterizes those trace links that have the same or similar structure (syntax) and/or purpose (semantics). For example, "implements", "tests", "refines" and "replaces" may be distinct trace link types. [13]*

Definition 6 (Traceability) *The potential for traces to be established and used. Traceability (i.e., trace "ability") is thereby an attribute of an artifact or of a collection of artifacts. [13]*

Definition 7 (Traceability Creation) *The general activity of associating two (or more) artifacts, by providing trace links between them. This can be done manually, automatically or semi-automatically, and additional annotations can be provided as desired to characterize attributes of the traces. [13]*

Definition 8 (Trace Recovery) *A particular approach to trace creation that implies the creation of trace links after[3] the artifacts that they associate have been generated and manipulated. These trace links may be created automatically or semi-automatically using tools. [13]*

2.3 Technological Background

This section provides background on key technologies used by this thesis. §2.3.1 introduces the MegaL modeling language and MegaL/Xtext as corresponding technology. §2.3.2 and §2.3.3 provides an introduction to JAXB and Hibernate respectively. §2.3.4 summarizes the relevant aspects of ANTLR (Another Tool For Language Recognition).

[3] The opposite to trace recovery is **Trace Capture**: *"A particular approach to trace creation that implies the creation of trace links concurrently with the creation of the artifacts that they associate. These trace links may be created automatically or semi-automatically using tools."*[13]. This is similar to the distinction between static program analysis and dynamic program analysis, thus leaving the scope of this thesis.

2.3.1 MagaL & MegaL/Xtext

MegaL [20] [9] is a modeling language for linguistic architectures developed by Software Languages Team[4] at the University of Koblenz-Landau. MegaL/Xtext [14] [**HaertelHHLV17**] is the integration of MegaL into the eclipse IDE (Integrated Development Environment) based on Xtext[5] developed by the same team.

MegaL provides a textual notation for typed ER Models, i.e. each entity and each relation is strictly typed. This is exemplified by listing 2.1 showing a simple megamodel for citation of scientific papers.

Listing 2.1 A Megamodel for Citation

```
Artifact < Entity
File < Artifact
Paper < File

cites < Paper * Paper

paper1 : Paper
paper2 : Paper
paper1 cites paper2
```

The types, starting with upper-case letters, state that there are files, which are papers. There is a relation `cites`, denoted with a type signature, stating one paper may cite the other. Eventually, there are two concrete paper entity instances `paper1` and `paper2`, where the former cites the latter instance.

MegaL/Xtext provides means for applying additional functionality or semantics to megamodels, e.g. for reasoning or validation on the induced entity graph. For this, MegaL/Xtext implements a plug-in system, which is exemplified with the following listings 2.2, 2.3 and 2.4.

Listing 2.2 MegaL/Xtext Plugin Integration

```
MyType < Entity

@Plugin 'MyRelationPlugin'
myRelation < MyType * MyType

MyRelationPlugin : Plugin
MyRelationPlugin = 'classpath:org.softlang.megal.MyRelationPlugin'
```

Listing 2.3 MegaL/Xtext GuidedReasonerPlugin base class

```
public abstract class GuidedReasonerPlugin extends InjectedReasonerPlugin {
  ...
  protected void guidedDerive(EntityType entityType) throws Throwable {}
  protected void guidedDerive(RelationshipType relationshipType) throws Throwable {}
```

[4] `http://www.softlang.org/`(Retrieved 12th December, 2017)

[5] `http://www.eclipse.org/Xtext/`(Retrieved 12th December, 2017)

```
    protected void guidedDerive(Entity entity) throws Throwable {}
    protected void guidedDerive(Relationship relationship) throws Throwable {}
}
```

Listing 2.4 MegaL/Xtext GuidedReasonerPlugin derived class

```
public class MyRelationPlugin extends GuidedReasonerPlugin {
    protected void guidedDerive(Relationship relationship) throws Throwable {
        ...
    }
}
```

Given a Java class artifact `MyRelationPlugin.java` implementing special features provided by the MegaL/Xtext API, e.g. the `GuidedReasonerPlugin` class, it can be bound to an entity instance by using its classpath. This plugin entity can then be used with the `@Plugin` annotation and applied to MegaL elements, i.e. entity types, relationship types, entity instances and relationship instances. `MyRelationPlugin` will be executed during evaluation of the megamodel.

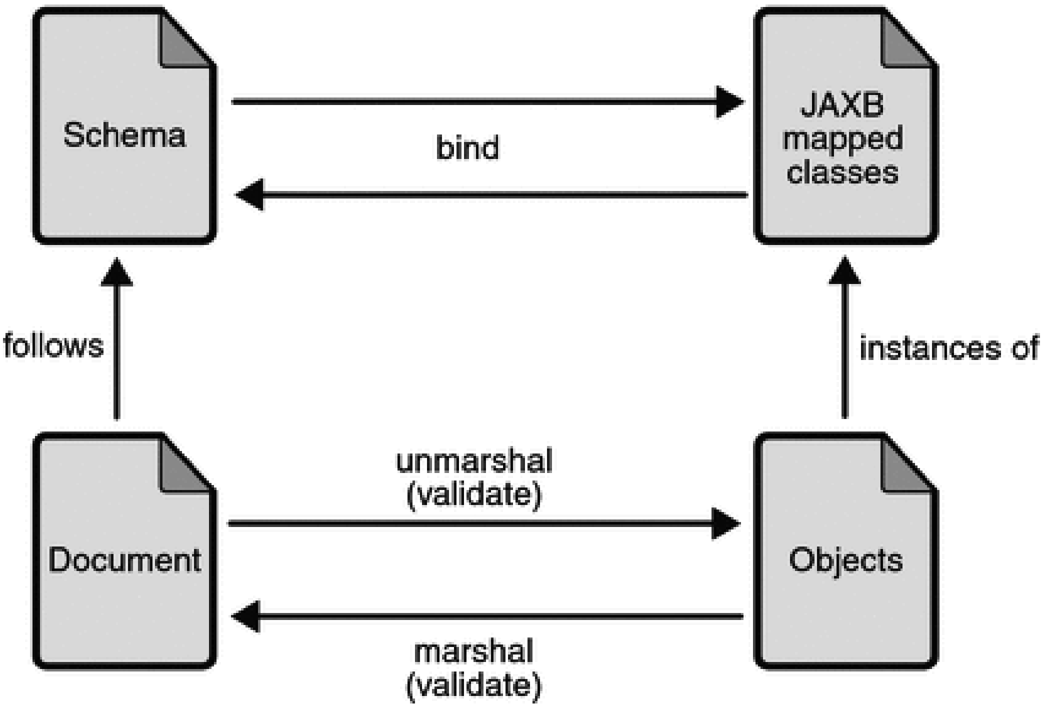
2.3.2 JAXB (Java Architecture for XML Binding)

JAXB [6] [8] is a Java technology developed by the Oracle Corporation for XML serialization and de-serialization of classes and instances. It implements O/X-Mapping (Object-XML-Mapping), i.e. Java classes can be transformed to XML schema files (XSD files) and instances of classes can be transformed to XML documents. Transformations can also be applied vice-versa. Figure 2.2 shows the O/X-Mapping facilitated by JAXB. The mapping of classes to schemas is called *binding*, whereas the mapping of instances to documents is called *marshaling* or *unmarshaling* respectively.

Binding and marshaling is defined through Java annotations. For instance, class and field declarations can be annotated to specify marshaling into specific syntactic XML elements, i.e. element tags or attributes. This is exemplified by listing 2.5:

Listing 2.5 A JAXB annotated class

```
@XmlRootElement(name="employee")
@XmlAccessorType(XmlAccessType.FIELD)
public class Employee {
    @XmlAttribute private int id;
    ...
}
```



This image is taken from [7]:
`https://docs.oracle.com/javase/tutorial/figures/jaxb/jaxb-dataBindingProcess.gif`
(retrieved 11th December, 2017)

Figure 2.2 Steps in the JAXB Binding Process

Annotations can define binding in more detail than shown above, e.g. it is possible to explicitly provide names and types of XML elements. Marshaling itself is triggered using reflection and the `JAXBContext` class:

Listing 2.6 JAXB marshaling in Java

```java
JAXBContext context = JAXBContext.newInstance(Employee.class);
context.createMarshaller().marshal(jaxbElement, writer);
```

More excessive examples and tutorials on JAXB can be found in [6] and in [8].

2.3.3 Hibernate

Hibernate [16] is a Java technology developed by Red Hat Inc. facilitating O/R-Mapping (Object-Relational-Mapping). Using Hibernate, one can define mappings from Java classes and their instances to tables of relational databases and their rows, and vice versa. This allows the underlying framework to generate SQL (Structured Query Language) code for communicating with a relational database system in order to provide means for CRUD operations.

Mappings can be defined using JPA (Java Persistence API) annotations or, traditionally, using XML files, i.e. for each Java class file `MyEntity.java` representing a data entity exists a mapping file `MyEntity.hbm.xml` describing the relational mapping. This is exemplified by the following two listings 2.7 and 2.8:

Listing 2.7 Excerpt of the Employee Java class

```
public class Employee {
   private int id;
   ...
}
```

Listing 2.8 Excerpt of the Employee Hibernate mapping file

```
<class name="Employee" table="EMPLOYEES">
   <id name="id" column="EMPLOYEE_ID">
       ...
   </id>
     ...
</class>
```

Here, the `Employee` class is mapped to the `EMPLOYEE` table its `id` field is mapped to the `EMPLOYEE_ID` column as primary key for the table. More excessive examples and tutorials on Hibernate can be found in [16].

2.3.4 ANTLR (Another Tool For Language Recognition)

ANTLR [27] is a technology providing means for processing structured text using parsing. It is available for Java, C#, JavaScript, Python and many other platforms [25], i.e. ANTLR can generate $LL(*)$ parsers for context-free languages in the target platform's language. Parsers are generated from an EBNF-like grammar notation, e.g. the following "hello"-grammar in listing 2.9 taken from [26]:

Listing 2.9 The Hello ANTLR grammar

```
grammar Hello;
hello   : 'hello' ID ;       // match keyword hello followed by an identifier
ID : [a-z]+ ;                // match lower-case identifiers
WS : [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines
```

ANTLR also provides means for integrating generated parsers into one's project, i.e. means for working with/traversing created Parse Trees. Tree traversal in order to transform Parse Trees to ASTs can be either done traditionally using the

Visitor Pattern [10] or using the combination of ANTLR Parse Tree walkers and walker listeners, a variation of the Observer Pattern [10].

Listener interfaces are generated during parser generation. All listeners implement the `ParseTreeListener` interface provided by the ANTLR runtime. The interface is shown in 2.10.

Listing 2.10 The ANTLR ParseTreeListener interface

```java
public interface ParseTreeListener {
   void visitTerminal(TerminalNode node);
   void visitErrorNode(ErrorNode node);
   void enterEveryRule(ParserRuleContext ctx);
   void exitEveryRule(ParserRuleContext ctx);
}
```

Listing 2.11 shows the generated listener interface for the grammar above.

Listing 2.11 An extension of the ANTLR ParseTreeListener interface

```java
public interface HelloListener extends ParseTreeListener {
    void enterHello(HelloContext ctx);
    void exitHello(HelloContext ctx);
}
```

While traversing a Parse Tree, listeners are notified upon entering or exiting tree nodes, called "context" by ANTLR. Traversal is triggered using `ParseTreeWalker` instances, which implement Depth-First Search (DFS). Usage of `ParseTreeWalker` is exemplified in listing 2.12.

Listing 2.12 Application of the ANTLR ParseTreeWalker

```java
...
HelloListener helloListener = ...
ParseTreeWalker parseTreeWalker = new ParseTreeWalker();
parseTreeWalker.walk(helloListener, parseTree);
...
```

An excessive introduction to ANTLR can be found in [27], a simple tutorial can be found in [26].

Chapter 3

Related Work

This chapter provides an non-exhaustive overview over related work regarding trace link recovery approaches. §3.1 summarizes selected papers. §3.2 provides some concluding remarks regarding the summarized approaches and the approach used by this thesis.

3.1 Paper Summary

This section summarizes selected papers regarding their approaches on trace link recovery.

Information Retrieval Methods for Automated Traceability Recovery
Authors in [21] propose a semi-automatic approach for recovering trace links: (i) corpus generation using document parsing to extract artifacts of a desired size (paragraphs, classes, methods, etc.); (ii) corpus indexing using information retrieval methods in order to create a homogeneous corpus removing stop words and applying other normalization techniques; (iii) recovery of "candidate" links computing a probabilistic similarity; (iv) analysis/assessment of candidate links conducted by a human removing false positive links and confirming correct links. It is argued that using probabilistic information retrieval methods reduces the overhead created by development of language specific analysis solutions, e.g. AST generation.

Using Rules for Traceability Creation Authors in [32] propose a fully automatic approach for recovering trace links by normalizing to XML documents and applying XQuery encoded traceability rules on the corpus. This approach also takes ambiguity of natural languages into account by applying a grammatical classification of sentences during preprocessing and considering synonyms during recovery.

Tracing object-oriented code into functional requirements Authors in [3] propose an information retrieval approach for recovering trace links between source code and natural language documentation artifacts. In a provided case-study, Java classes are linked to requirement documents based on a probabilistic similarity computed from the set of identifiers found in a class artifact.

Recovering Code to Documentation Links in OO Systems Authors in [2] propose an information retrieval approach for recovering trace links between source code and natural language documentation artifacts based on a probabilistic similarity computed from the set of identifiers found in source code artifacts. This is a predecessor paper of [3].

Mining software repositories for traceability links Authors in [17] propose an approach for recovering trace links among source code and informal documentation artifacts by monitoring their co-change frequency, i.e. artifacts that are changed to concurrently in the history of a version control system. It is argued that a high co-change frequency implies trace links among artifacts.

Combining Textual and Structural Analysis of Software Artifacts for Traceability Link Recovery Authors in [23] propose a combined approach of information retrieval methods for recovering exogenous links between source code and documentation artifacts and classical analysis methods for recovering endogenous links among source code artifacts. Endogenous links among documentation artifacts are proposed to be recovered indirectly, i.e. if source artifacts $S1$ and $S2$ are linked and both ar linked to documentation artifacts $D1$ and $D2$ respectively, then this may be seen as evidence that $D1$ are also linked $D2$. Indirect linking is proposed because the authors argue that information retrieval methods alone are not sufficient for trace link recovery purposes. Documentation artifacts may be linked despite sharing little to no semantic similarities.

Recovering traceability links in software artifact management systems using information retrieval methods Authors in [22] provide an excessive case-study evaluating information retrieval methods, namely Latent Semantic Indexing, for trace link recovery. It concludes that information retrieval is not sufficient to recover all links among source code artifacts,

TraceME: Traceability Management in Eclipse Authors in [5] introduce TraceME, an eclipse IDE base tool for trace link recovery and management. It utilizes information retrieval techniques, namely the Vector Space Model, for trace link recovery inside eclipse projects.

OpenTrace: An Open Source Workbench for Automatic Software Traceability Link Recovery Authors in [1] introduce OpenTrace, a workbench for reproducible experiments with focus on automated trace link recovery. It provides means to apply information retrieval techniques for recovering trace links to large corpora of natural language artifacts.

Traceclipse: An eclipse plug-in for traceability link recovery and management Authors in [18] introduce Traceclipse, another eclipse IDE based tool, providing means for trace link recovery using information retrieval techniques, namely the Vector Space Model. The implemented recovery process is semi-automatic in the sense that a user has to inspect proposed links.

3.2 Remarks

The majority of the papers summarized in §3.1 utilize information retrieval techniques for trace link recovery [32] [3] [2] [23] [22] [5] [1] [18] The approach we use for this thesis is to utilize static program analysis techniques for trace link recovery. This notable difference is due to the differing source and target artifacts intended to be recovered. The summarized approaches intend to recover links between source code and informal, natural language artifacts used for documentation. Because of the latter, applying information retrieval methods seems to be a reasonable strategy. Only [23] argues information retrieval to be insufficient. However, the proposed recovery method (coupling metrics) for inter source code trace links seem to assume a homogeneous corpus. This thesis aims to recover exogenous trace links in a heterogeneous corpus.

Chapter 4

Design

This chapter summarizes the design of the recovery system developed for this thesis. §4.1 summarizes functional requirements of the developed system. §4.2 recapitulates design of the recovery system with respect to the specified requirements. §4.3 summarizes integration in the recovery system with the MegaL/Xtext environment .

4.1 Requirements

This section summarizes functional requirements for the recovery system developed as part of this thesis. All presented requirements are must haves, so no priority differentiation is applied.

Requirement 1 *Fragment Recovery*. The recovery system has to recover fragments, i.e. syntactically well-formed parts of artifacts (see §2.1.2).

Requirement 2 *Parthood Recovery*. The recovery system has to recover parthood links of fragments (see §2.1.1 and §2.1.2)).

Requirement 3 *Correspondence Recovery*. The recovery system has to recover correspondence links, i.e. links capturing the relation between fragments of artifacts denoting both represent the same data (see §2.1.3).

Requirement 4 *Conformance Recovery*. The recovery system has to recover conformance links, i.e. links capturing the relation between artifacts or fragments denoting that one defines the other (see §2.1.4).

Requirement 5 *MegaL/Xtext Integration.* The recovery system has to run within MegaL/Xtext, i.e. the implementations of requirements 1, 2, 3 and 4 must be compatible and integrated with the MegaL/Xtext plug-in-system.

4.2 Recovery Design

This section summarizes the design of the recovery system developed for this thesis. §4.2.1 will describe the all over process of the recovery system. §4.2.2 will describe the design core API and its components developed for the recovery system. §4.2.3 will describe the design of the actual system for recovering links among O/R/X-Mapping artifacts.

4.2.1 Recovery Process

The recovery process is a straight forward analysis of two artifacts. Figure 4.1 shows a flowchart depicting this. Given two artifacts as input, the recovery process works as follows:

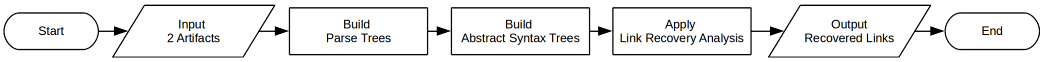


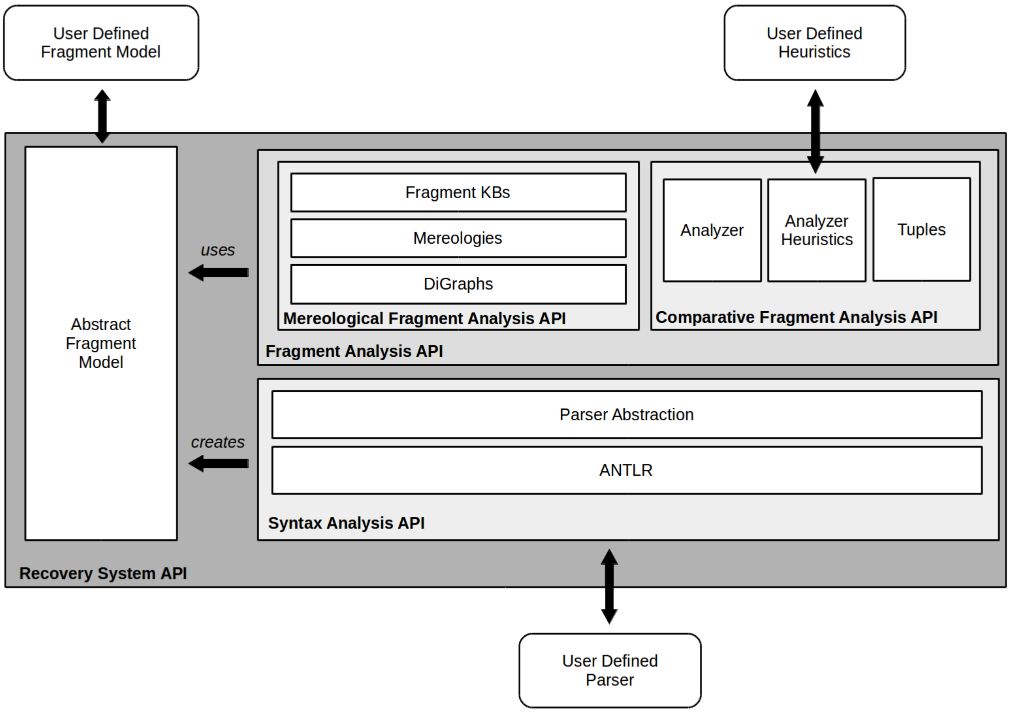
Figure 4.1 The Recovery Process

1. From each artifact a Concrete Syntax Tree (CST) or Parse Tree is constructed,
2. each Parse Tree is further refined into an AST,
3. both ASTs are compared with each other, i.e. both trees are traversed in DFS fashion and each pair of nodes is checked whether it can be recovered as link.

Eventually, the set of recovered links serves as output of the process.

4.2.2 Recovery API

The Recovery API is the core of the developed recovery system. It provides generalized data structures and methods for syntactic analysis of artifacts and their



This block diagram depicts the functional outline of the Recovery System API (note, that this does not necessarily correspond to the package outline of its actual implementation).

Figure 4.2 The Recovery API

fragments. Figure 4.2 depicts the API as block diagram. The components of the API are:

Abstract Fragment Model The Abstract Fragment Model serves as base model for all ASTs. It can be thought of as Data Transfer Object (DTO) for the analysis components. As user of the API, I have to derive a specific AST from this model. A detailed description follows in §4.2.2.1.

Syntax Analysis API The Syntax Analysis API is the abstraction layer for parsing and AST construction. It is currently backed by ANTLR, but its internal design is loosely coupled, so other parser libraries can be used. As user of the API, I have to implement a parser constructing an AST deriving the Abstract Fragment Model. However, if one uses ANTLR, only AST construction from a CST is required. A detailed description follows in §4.2.2.2.

Fragment Analysis API The Fragment Analysis API consists of two components:

Mereological Fragment Analysis API The Mereological Fragment Analysis API provides components for deriving parthood links between syntactically well-formed fragments. As user of the API, I only have to apply its components to constructed ASTs. A detailed description follows in §4.2.2.3.

Comparative Fragment Analysis API The Comparative Fragment Analysis API provides components for deriving links between different artifacts and their fragments. As user of the API, I have to implement one or more specialized heuristics for deciding which links can be recovered. A detailed description follows in §4.2.2.4.

4.2.2.1 Abstract Fragment Model

The Abstract Fragment Model describes a tree in which each node represents an syntactically well-formed fragment. Figure 4.3 shows an UML (Unified Modeling Language) class diagram of the model. The resulting tree data structure is doubly-linked, i.e. an `IFragment` node aggregates references to its children an to its parent, given it is not the root node. Each fragment `IFragment` contains the text it represents. In order to distinguish fragments which represent the same text, each node also carries the text's position in the artifact through `IFragmentPosition` instances. Positions inside the text are determined by the start and ending line number as well as the corresponding first and last character inside the line. Most of `IFragment`'s relevant code for trees is pre-implemented in the `BaseFragment` abstract class.

4.2.2.2 Syntax Analysis API

The Syntax Analysis API provides abstraction for AST construction. The API itself is really small, it only consists of the `IParser` and `IParserFactory` interfaces. However, its implementation for ANTLR is designed to reduce ANTLR-specific boilerplate code. Figure 4.4 shows the UML class diagram for the ANTLR backed implementation. One can see, that this implementation makes heavy use of the Abstract Factory Pattern [10]. This allows for a quick and easy definition of new parsers as shown in listing 4.1.

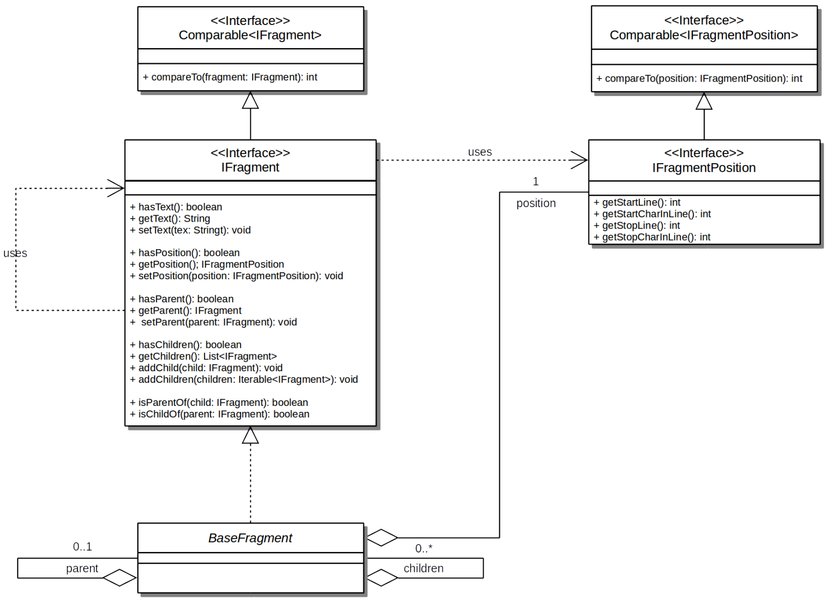

<<Interface>>
Comparable<IFragment>
+ compareTo(fragment: IFragment): int
<<Interface>>
Comparable<IFragmentPosition>
+ compareTo(position: IFragmentPosition): int
<<Interface>>
IFragment
+ hasText(): boolean
+ getText(): String
+ setText(tex: Stringt): void
+ hasPosition(): boolean
+ getPosition(); IFragmentPosition
+ setPosition(position: IFragmentPosition): void
+ hasParent(): boolean
+ getParent(): IFragment
+ setParent(parent: IFragment): void
+ hasChildren(): boolean
+ getChildren(): List<IFragment>
+ addChild(child: IFragment): void
+ addChildren(children: Iterable<IFragment>): void
+ isParentOf(child: IFragment): boolean
+ isChildOf(parent: IFragment): boolean
uses
uses
1
position
<<Interface>>
IFragmentPosition
+ getStartLine(): int
+ getStartCharInLine(): int
+ getStopLine(): int
+ getStopCharInLine(): int
BaseFragment
0..1
parent
0..*
children

Figure 4.3 The Abstract Fragment Model

Listing 4.1 IParserFactory Usage Example

```
...
IParser java8Parser = parserFactory.newParser(
    Java8Lexer::new,
    Java8Parser::new,
    Java8Parser::compilationUnit,
    Java8FragmentBuildingListener::new);
...
```

Listing 4.1 demonstrates the creation of a new parser using an `IParserFactory` instance with Java8 features. Lexer and Parser classes are generated by ANTLR. A suitable listener has to be implemented.

The other advantage of the Abstract Factory Pattern is that the instantiation of all relevant classes necessary for the creation of ANTLR Parse Trees takes place in the predefined `AntlrParser` class. This is exemplified by listing 4.2.

Listing 4.2 AntlrParser ParseTree Creation

```
...
CharStream charStream = antlrCharStreamFactory.newCharStream(inputStream);
Lexer lexer = antlrLexerFactory.newLexer(charStream);
TokenStream tokenStream = antlrTokenStreamFactory.newTokenStream(lexer);
Parser parser = antlrParserFactory.newParser(tokenStream);
```

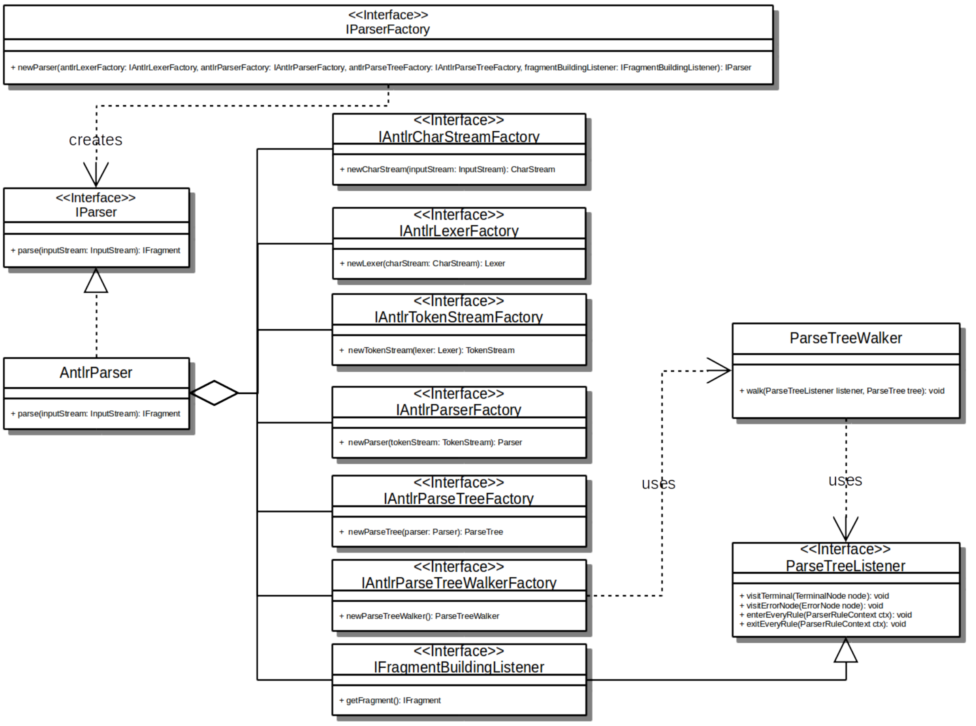

<<Interface>>
IParserFactory
+ newParser(antlrLexerFactory: IAntlrLexerFactory, antlrParserFactory: IAntlrParserFactory, antlrParseTreeFactory: IAntlrParseTreeFactory, fragmentBuildingListener: IFragmentBuildingListener): IParser
creates
<<Interface>>
IParser
+ parse(inputStream: InputStream): IFragment
AntlrParser
+ parse(inputStream: InputStream): IFragment
<<Interface>>
IAntlrCharStreamFactory
+ newCharStream(inputStream: InputStream): CharStream
<<Interface>>
IAntlrLexerFactory
+ newLexer(charStream: CharStream): Lexer
<<Interface>>
IAntlrTokenStreamFactory
+ newTokenStream(lexer: Lexer): TokenStream
<<Interface>>
IAntlrParserFactory
+ newParser(tokenStream: TokenStream): Parser
<<Interface>>
IAntlrParseTreeFactory
+ newParseTree(parser: Parser): ParseTree
<<Interface>>
IAntlrParseTreeWalkerFactory
+ newParseTreeWalker(): ParseTreeWalker
<<Interface>>
IFragmentBuildingListener
+ getFragment(): IFragment
uses
ParseTreeWalker
+ walk(ParseTreeListener listener, ParseTree tree): void
uses
<<Interface>>
ParseTreeListener
+ visitTerminal(TerminalNode node): void
+ visitErrorNode(ErrorNode node): void
+ enterEveryRule(ParserRuleContext ctx): void
+ exitEveryRule(ParserRuleContext ctx): void

Figure 4.4 The Syntax Analysis API

```
ParseTree parseTree = antlrParseTreeFactory.newParseTree(parser);
...
```

Listing 4.2 demonstrates the creation of an ANTLR `ParseTree` instance as implemented by the `AntlrParser` class.

Creation of ASTs or fragment trees is done using the `ParseTreeWalker` and `ParseTreeListener` infrastructure provided by ANTLR as described in §2.3.4. This is an variation of the Observer Pattern. Listeners in conjunction with walkers are an alternative to the Visitor Pattern provided by ANTLR. An instance of `ParseTreeWalker` traverses a Parse Tree using DFS. During traversal, a designated method of `ParseTreeListener` is executed. For AST creation, an API user has to implement the `IFragmentBuildingListener` interface, which is an extension of the `ParseTreeListener` interface. The creation of a fragment tree by `AntlrParser` is exemplified in listing 4.3.

Listing 4.3 AntlrParser Fragment Creation

```
...
ParseTreeWalker parseTreeWalker = antlrParseTreeWalkerFactory.newParseTreeWalker();
```

```
parseTreeWalker.walk(fragmentBuildingListener, parseTree);
IFragment fragment = fragmentBuildingListener.getFragment();
...
```

Listing 4.3 demonstrates the creation of an `IFragment` AST instance as implemented by the `AntlrParser` class.

4.2.2.3 Mereological Fragment Analysis API

The Mereological Fragment Analysis API provides components for deriving parthood links between syntactically well-formed fragments. Direct parthood links are recovered from parent/child relationships within an `IFragment` AST. However, because parthood is transitive, we also need to recover these links. This is done using a digraph data structure upon which we can compute its reflexive transitive closure.

Figure 4.5 shows an UML class diagram depicting the relevant classes and interfaces of the Mereological Fragment Analysis API. At its heart, the API utilizes the generic `IDiGraph` interface whose implementations provide means for interrelating comparable objects, i.e. `IFragment` instances.

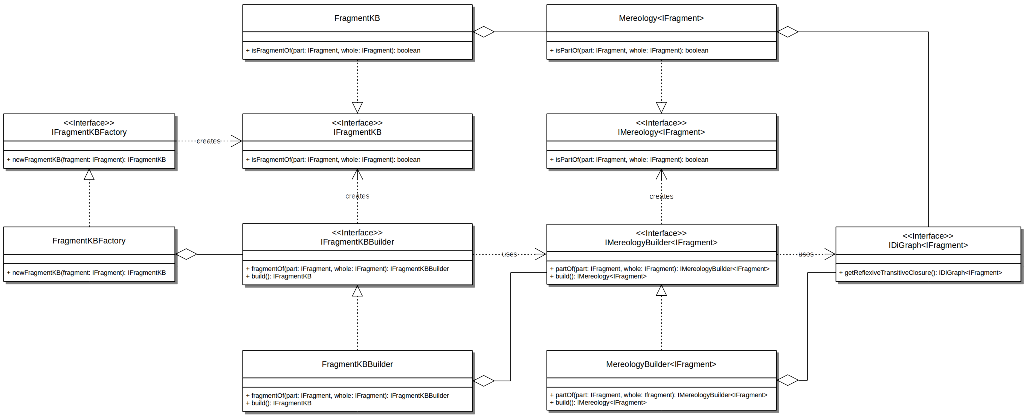

FragmentKB
+ isFragmentOf(part: IFragment, whole: IFragment): boolean
Mereology<IFragment>
+ isPartOf(part: IFragment, whole: IFragment): boolean
<<Interface>>
IFragmentKBFactory
+ newFragmentKB(fragment: IFragment): IFragmentKB
creates
<<Interface>>
IFragmentKB
+ isFragmentOf(part: IFragment, whole: IFragment): boolean
<<Interface>>
IMereology<IFragment>
+ isPartOf(part: IFragment, whole: IFragment): boolean
creates
creates
FragmentKBFactory
+ newFragmentKB(fragment: IFragment): IFragmentKB
<<Interface>>
IFragmentKBBuilder
+ fragmentOf(part: IFragment, whole: IFragment): IFragmentKBBuilder
+ build(): IFragmentKB
uses
<<Interface>>
IMereologyBuilder<IFragment>
+ partOf(part: IFragment, whole: IFragment): IMereologyBuilder<IFragment>
+ build(): IMereology<IFragment>
uses
<<Interface>>
IDiGraph<IFragment>
+ getReflexiveTransitiveClosure(): IDiGraph<IFragment>
FragmentKBBuilder
+ fragmentOf(part: IFragment, whole: IFragment): IFragmentKBBuilder
+ build(): IFragmentKB
MereologyBuilder<IFragment>
+ partOf(part: IFragment, whole: Ifragment): IMereologyBuilder<IFragment>
+ build(): IMereology<IFragment>

Figure 4.5 Mereological Fragment Analysis API

This digraph is wrapped by generic `IMereology` implementations, a data structure providing query methods on the topic of mereology, i.e. parthood relations. Mereologys are constructed using the Builder Pattern [10]. An `IMereology-`

Builder instance adds nodes and edges into its digraph with semantically named methods allowing a descriptive programming style.

IMereology's are then further wrapped by IFragmentKB (a Knowledge Base over IFragment) implementations in order to avoid dealing with generics throughout the system. This is also done utilizing the Builder Pattern.

The recovery of parthood links is encapsulated through IFragmentKBFactory using the Abstract Factory Pattern, which computes IFragmentKB instances from an IFragment AST as input.

4.2.2.4 Comparative Fragment Analysis API

The Comparative Fragment Analysis API provides components for deriving links from two IFragment ASTs generated from different artifacts. Figure 4.6 shows an UML class diagram depicting the relevant interfaces and classes of the API. It utilizes the Strategy Pattern [10] through the IFragmentAnalyzerHeuristic interface since concrete behavior of the recovery analysis is for the user to implement. In this context, strategies are called heuristics, because recovery of links is not necessarily based on optimal, i.e. absolutely correct, solutions. Instead strategies may also implement practical solutions with reasonably sufficient results.

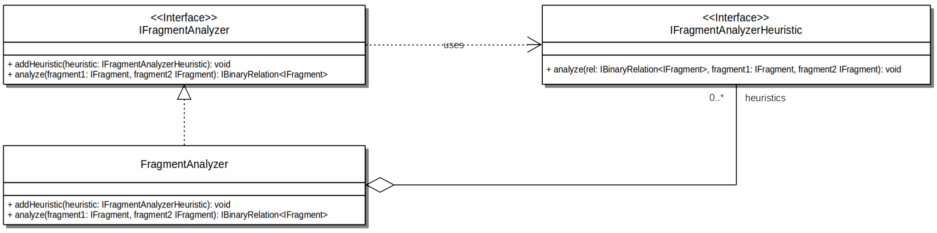

<<Interface>>
IFragmentAnalyzer
+ addHeuristic(heuristic: IFragmentAnalyzerHeuristic): void
+ analyze(fragment1: IFragment, fragment2 IFragment): IBinaryRelation<IFragment>
uses
<<Interface>>
IFragmentAnalyzerHeuristic
+ analyze(rel: IBinaryRelation<IFragment>, fragment1: IFragment, fragment2 IFragment): void
0..*
heuristics
FragmentAnalyzer
+ addHeuristic(heuristic: IFragmentAnalyzerHeuristic): void
+ analyze(fragment1: IFragment, fragment2 IFragment): IBinaryRelation<IFragment>

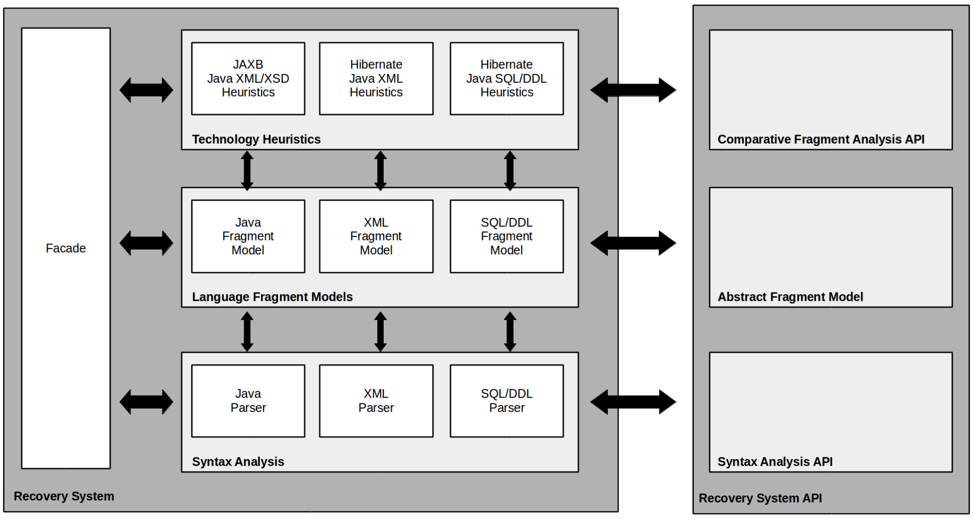
Figure 4.6 Comparative Fragment Analysis API

IFragmentAnalyzer allows to apply multiple heuristics, since there may be more than one practical approach. This also allows strategies to keep a relatively small implementation footprint.

Recovered links are captured and stored in an IBinaryRelation instance which works like a set of pairs.

4.2.3 Recovery System

The Recovery System is the actual implementation of parthood, correspondence and conformance link recovery for Java based O/R/X-Mapping artifacts. Figure 4.7 shows a block diagram outlining the functional components of the system.



This block diagram depicts the functional outline of the Recovery System and its dependencies to the Recovery System API (note, that this does not necessarily correspond to the package outline of its actual implementation).

Figure 4.7 The Recovery System

The Recovery System utilizes the Syntactic Analysis API described in §4.2.2.2 for Java, XML and SQL/DDL. For this, the Abstract Fragment Model described in §4.2.2.1 is implemented. Note, fragment models do not implement an AST suitable for compilation of the targeted language. Syntactic features unnecessary for the intended analysis, like local variable declarations, arithmetic expressions or invocations, are omitted. Fragment models rather focus on structural features of the languages at hand.

The Comparative Fragment Analysis API described in §4.2.2.4 is implemented with focus on artifacts of technologies, namely JAXB and Hibernate. It implements heuristics for link recovery between:

- Java and XML for JAXB artifacts, i.e. Java models are serialized as XML

- Java and XSD for JAXB artifacts, i.e. Java models are serialized as XSD
- XML and XSD for JAXB artifacts, i.e. the two previous scenarios occurred
- Java and XML for Hibernate mapping artifacts, i.e. Hibernate uses XML meta-data for O/R-Mapping.
- Java and SQL/DDL for Hibernate generated SQL artifacts, i.e. Hibernate uses Java annotations for O/R-Mapping

On top of the actual implementation, the Recovery System also utilizes the Facade Pattern [10]. This is to provide a single access point for all implemented analysis features.

4.3 Megal/Xtext Integration Design

This section summarizes the design of the recovery system's integration in the MegaL/Xtext environment. Figure 4.8 shows a block diagram depicting the outline of this integration. Note, the recovery system is implemented as separate project, which is then included as reference (through a `.jar` file) in a MegaL/Xtext project.

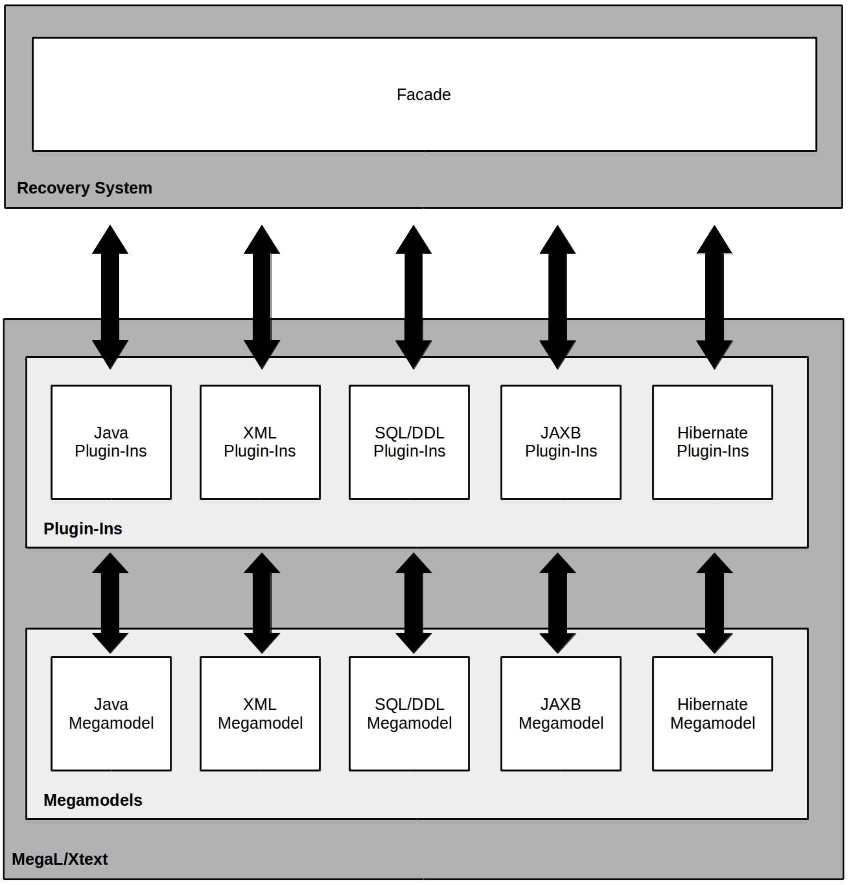
MegaL/Xtext provides a plug-in system, which allows to bind special Java classes to `Plugin` entities (see §2.3.1). Code of such classes is then executed during the evaluation of a megamodel. Listing 4.4 exemplifies the declaration of plug-ins within MegaL.

Listing 4.4 MegaL/Xtext Plugin Integration

```
...
JavaFragmentRecoveryPlugin : Plugin
JavaFragmentRecoveryPlugin realizationOf Java
JavaFragmentRecoveryPlugin partOf FileFragmentRecoveryReasonerPlugin
JavaFragmentRecoveryPlugin = 'classpath:org.softlang.megal.plugins.impl.java.JavaFragmentRecoveryPlugin'
...
```

The snippet in listing 4.4 demonstrates the instantiation of `JavaFragmentRecoveryPlugin` as part of `FileFragmentRecoveryReasonerPlugin` in MegaL/Xtext.

The Recovery System or rather its facade (see §4.2.3) is used to implement MegaL/Xtext plug-ins. These plug-ins are then in turn bound within megamodels, which are divided by language and technology, i.e. there are separate MegaL modules for Java, XML, SQL, Hibernate and JAXB.



This block diagram depicts the functional outline of the Recovery System's integration into the MegaL/Xtext environment (note, that this does not necessarily correspond to the outline of its actual implementation).

Figure 4.8 Integration of the Recovery System int MegaL/Xtext

Chapter 5

Implementation

This chapter summarizes the implementation for crucial parts of the recovery system implemented for this thesis. §5.1 covers the implementation of fragment and parthood link recovery. §5.2 covers the implementation of correspondence and conformance link recovery

5.1 Recovering Fragments & Parthood Links

This section summarizes the implementation of fragment recovery. Fragments are syntactically well-formed pieces of code. For instance, consider the Java class in listing 5.1.

Listing 5.1 Excerpt of the Company Java class

```
public class Company {
   ...
   private String name;
   ...
   public String getName() {
      return name;
   }
   ...
}
```

It consists, among others, of the following fragments:

- a field declaration fragment for `name`:

  ```
  private String name;
  ```

- a method declaration fragment for an accessor-method of `name`:

```
public String getName() {
   return name;
}
```

- the return statement of the accessor-method:

  ```
  return name;
  ```

- the class declaration itself can also be considered a fragment.

The task of recovering fragments is to add entities for each of such code pieces into a megamodel. Figure 5.1 exemplifies the recovery of Java[1] fragments in an idealized form:

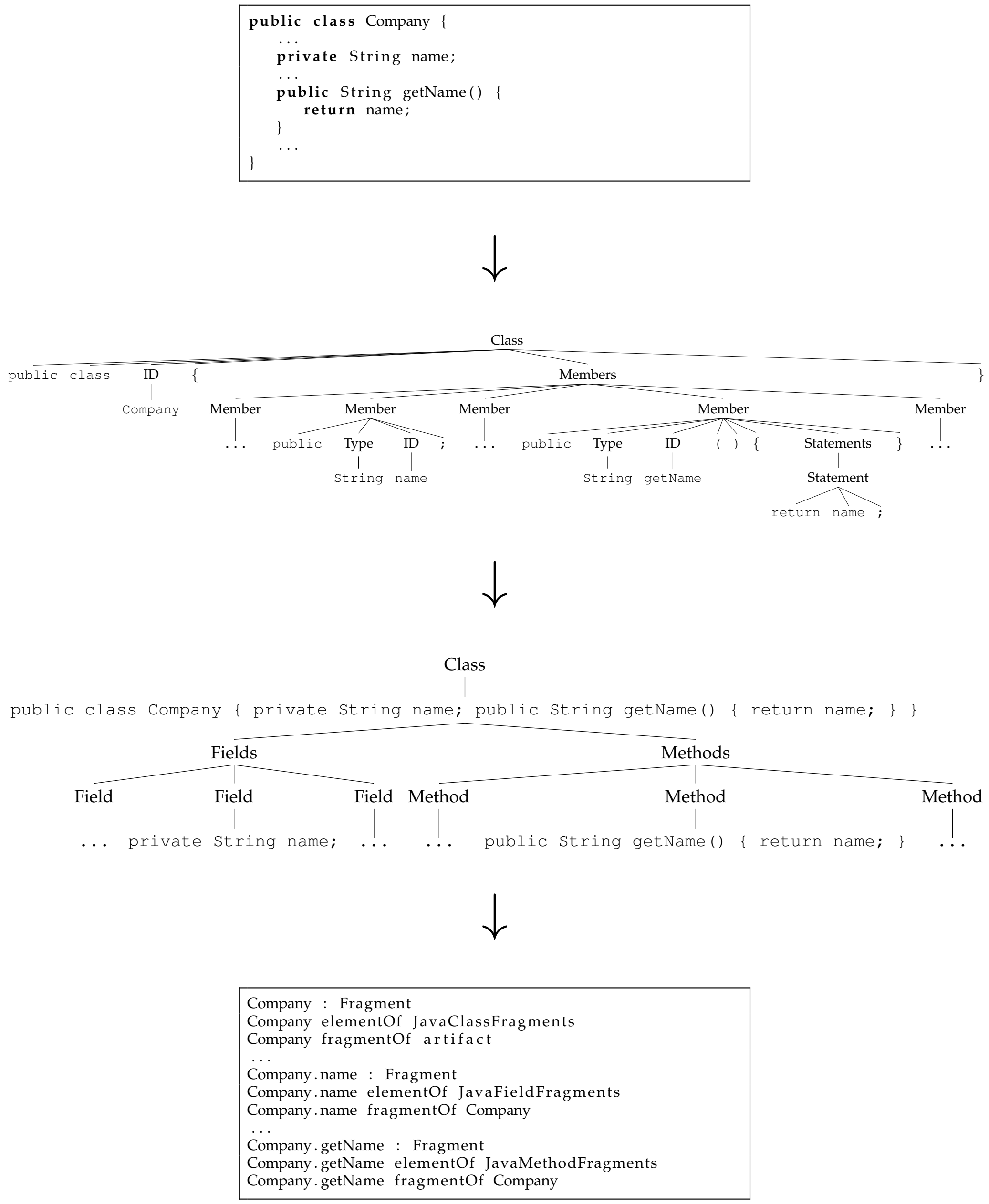
1. a Parse Tree is generated from an artifact; this is done via ANTLR as described in §4.2.2.2 and does not require further explanation, since it is only an application of the most common ANTLR use case
2. then the Parse Tree is transformed into an concrete fragment AST model; this is also done relying on ANTLR tree traversal utilities and described in detail in §5.1.1
3. eventually, nodes of the generated fragment AST are added to a megamodel as entities; the details are described in §5.1.2.

5.1.1 Concrete Fragment Models

As mentioned in §4.2.2.1, Concrete Fragment Models are derivations of the Abstract Fragment Model of the Recovery System API. Figure 5.2 shows an UML class diagram of the implemented Fragment AST for Java. All fragment classes derive from `IFragment` through the base class `JavaFragment`. From here, several specializations are introduced:

- `IdentfiedJavaFragment` for constructs with identifiers
- `ModifiedJavaFragment` for constructs with modifiers like `private, public, final, abstract, static`, etc. or annotation meta-data

[1] Recovery for XML and SQL/DDL is implemented in a similar fashion. If there is a noteworthy difference for other languages it will be explored, otherwise we keep using Java as example domain for the remaining sections of §**??**.



```
public class Company {
   ...
   private String name;
   ...
   public String getName() {
      return name;
   }
   ...
}
```

```
Company : Fragment
Company elementOf JavaClassFragments
Company fragmentOf artifact
...
Company.name : Fragment
Company.name elementOf JavaFieldFragments
Company.name fragmentOf Company
...
Company.getName : Fragment
Company.getName elementOf JavaMethodFragments
Company.getName fragmentOf Company
```

This picture shows an idealized recovery of Java fragments:

code artifact → Parse Tree → fragment AST → megamodel

Both Parse Tree and AST are depicted in a simplified, schematic form.

Figure 5.1 Idealized Recovery of Java Fragments

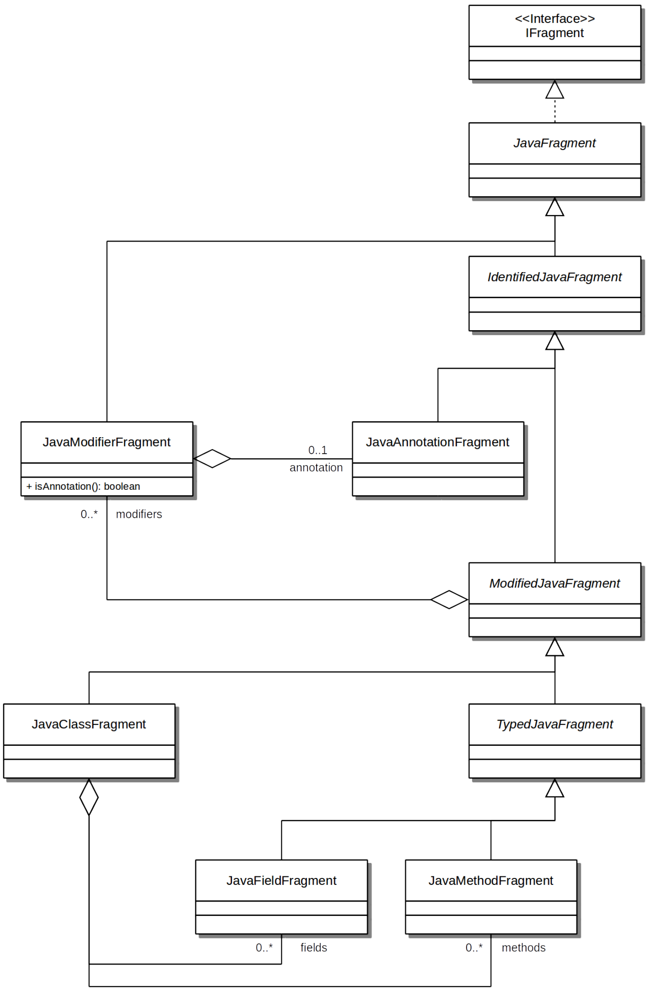


Figure 5.2 Java Fragment AST Model

- `TypedJavaFragment` for constructs with a distinct (return-) type like fields, methods or variables

The fragment model for Java only focuses on structural syntactical features of the language. Everything below method signature level is omitted, because fragments of this level are not important for the scope of this thesis. On the other hand, annotations are captured, since they provide significant information for further recovery analysis. Hibernate and JAXB rely on annotations for O/R/X-Mapping.

The AST is constructed using the ANTLR `ParseTreeListener` approach as described in §4.2.2.2.

Listing 5.2 Construction of Java Fragment ASTs

```
...
@Override
public void exitMethodDeclaration(Java8Parser.MethodDeclarationContext ctx) {
    javaMethodFragments.push(java8FragmentFactory.newJavaMethodFragment(ctx, javaMethodModifierFragments));
}

@Override
public void exitNormalClassDeclaration(Java8Parser.NormalClassDeclarationContext ctx) {
    javaClassFragments.push(java8FragmentFactory.newJavaClassFragment(ctx, javaClassModifierFragments,
            javaFieldFragments, javaMethodFragments, declaredPackage));
}
...
```

Listing 5.2 shows an excerpt from the listener implementation used to construct the the AST for Java fragments. Here, fragment instances are created when the listener leaves certain nodes of the Parse Tree. Creation is delegated to a factory. The listener pushes fragments onto stacks, making the overall implementation work like a pushdown automaton or stack machine.

5.1.2 Megamodeling Fragments

In order to add fragments into a megamodel, a suitable linguistic model has to be declared first. Consider the following instance level megamodel of recovered fragments shown in listing 5.3.

Listing 5.3 A Megamodel for fragments of a Java class

```
File < Artifact
Fragment < Artifact
...
elementOf < Set * Set
fragmentOf < Fragment * Artifact
...
aJavaFile : File
aJavaFile = 'Company.java'
...
aJavaFile.Company : Fragment
aJavaFile.Company elementOf JavaClassFragments
aJavaFile.Company fragmentOf aJavaFile
...
aJavaFile.Company.name : Fragment
aJavaFile.Company.name elementOf JavaFieldFragments
aJavaFile.Company.name fragmentOf aJavaFile
aJavaFile.Company.name fragmentOf Company
...
aJavaFile.Company.getName : Fragment
aJavaFile.Company.getName elementOf JavaMethodFragments
aJavaFile.Company.getName fragmentOf aJavaFile
aJavaFile.Company.getName fragmentOf Company
```

This megamodel shows fragments of a Java file `Company.java`. It contains entities denoting a class declaration, a field declaration and a method declaration fragment. All are declared instances of the entity type `Fragment`. Entity names use dot-notation for implying parthood, i.e.:

$$a.b \Rightarrow b \textsf{ partOf } a$$

Further details of the naming-scheme employed for fragment entities are explored in §5.1.2.2.

We use `elementOf` for further specialization and/or generalization of the entitie's kind[2]. However, the right-hand side of these relationships used to specialize fragments are not simply sets, they are actually modeled as languages as we will see in §5.1.2.1.

5.1.2.1 Fragment Languages

A fragment alone cannot be element of the language the artifact it originated from belongs to, e.g. Java methods alone cannot be accepted by the Java grammar, nor can XML attributes alone be accepted by the XML grammar. Therefore, in order to cleanly add fragments into a megamodel, we need to model fragment languages. Listing 5.4 shows the megamodel for Java fragments.

Listing 5.4 A Megamodel for Java Fragment

```
...
Language < Set
...
fragmentLanguageOf < Language * Language
...
Java : Language

JavaFragments : Language
JavaFragments fragmentLanguageOf Java

JavaClassFragments : Language
JavaClassFragments subsetOf JavaFragments

JavaFieldFragments : Language
JavaFieldFragments subsetOf JavaFragments

JavaMethodFragments : Language
JavaMethodFragments subsetOf JavaFragments
...
```

[2] We use the term *kind* to avoid confusion with entity types, although we use it to refer to fragment types.

It introduces the special relationship type `fragmentLanguageOf`, denoting the left-hand side is the language containing all possible fragments the right-hand side's language elements can be deconstructed in. This also implies a super-set relation between the two languages as fragmentOf is reflexive like partOf, so:

$$L_1 \texttt{ fragmentLanguageOf } L_2 \Rightarrow L_2 \subseteq L_1$$

5.1.2.2 Fragment Entity Identifiers

MegaL entity identifiers have to be unique, but names of the constructs represented by fragment do not necessarily adhere to that. For instance, a Java method may be overloaded or multiple XML elements are naturally identified via the same name. Because of this, a naming scheme for storing recovered fragments in a MegaL megamodel needs to be devised.

We utilize a simple naming scheme, preserving both uniqueness and readability. Consider listing 5.5, showing an excerpt of a XML file.

Listing 5.5 Excerpt of a Company XML file

```
<company id="0" name="Softlang Inc.">
    <departments>
        <department id="0" name="Grammar Engineering">
          ...
        </department>
        <department id="1" name="Research &amp; Development">
            ...
        </department>
          ...
    </departments>
</company>
```

If we want to add both `department` fragments to a megamodel, we prepend an indexing prefix to its name, i.e. `F<index>$`. This is exemplified in listing 5.6.

Listing 5.6 Qualified Fragment Identifiers

```
xmlFile : File
xmlFile.company : Fragment
xmlFile.company.departments : Fragment
xmlFile.company.departments.F0$department : Fragment
xmlFile.company.departments.F1$department : Fragment
```

5.1.3 Recovering Parthood Links

This section summarizes the implementation of parthood link recovery. Recovering parthood links exploits the nature of Parse Trees and ASTs as mentioned

§4.2.2.3, where the nested structure of code is represented by parent/child association between nodes. Listing 5.7 shows recovery of parthood links from a tree data structure in pseudo code notation.

Listing 5.7 Pseudo code Recovery of Parthood Links

```
input: A node n of a n–ary tree structure.
procedure recoverParthoodLinks(n):
   foreach child node c of n:
      declare c partOf n.
      recoverParthoodLinks(c).
```

The recovery procedure utilizes recursive descent traversal on a tree. The actual implementation of parthood link recovery adds partOf- and fragmentOf-links simultaneously into a simple knowledge base backed by a graph data structure. Listing 5.8 shows the responsible methods for this.

Listing 5.8 Actual Recovery of Parthood Links

```
private IFragmentKBBuilder newFragmentKB(IFragmentKBBuilder fragmentKBBuilder, IFragment fragment) {
    for (IFragment child : fragment.getChildren()) {
        fragmentKBBuilder = newFragmentKB(fragmentKBBuilder.fragmentOf(child, fragment), child);
    }
    return fragmentKBBuilder;
}

@Override
public IFragmentKB newFragmentKB(IFragment fragment) {
   return newFragmentKB(fragmentKBBuilderFactory.newFragmentKBBuilder(), fragment).build();
}
```

Once a graph of recovered partOf- and/or fragmentOf-links is present, we can compute its transitive closure, thus recovering transitive partOf- and fragmentOf-links. Listing 5.9 shows the pseudo code for computing the transitive closure of a graph.

Listing 5.9 Pseudo code Recovery of transitive Parthood Links

```
input: A graph g and vertex v.
procedure reachableVertices(g,v):
   mark v as walked.
   foreach adjacent vertex w of v:
      if w is not marked as walked:
         yield w.
         reachableVertices(g,w).

input: A graph g.
procedure computeTransitiveClosure(g):
   foreach vertex v of g:
      foreach vertex w in reachableVertices(g,v):
         if w != v and g does neither contain edge (v,w) nor (w,v):
            add edge (v,w) to g.
```

Note, the pseudo code implementation for computation of transitive closures uses a recursive DFS for computing transitively reachable vertices of a given vertex. It utilizes a yield-return, denoting the next value is available from the outside without exiting the procedure. This can be thought of sequence which would be returned normally. The actual implementation utilizes the Iterator Pattern [10] and non-recursive DFS for computing transitively reachable vertices.

5.2 Recovering Correspondence & Conformance Links

This section summarizes the implementation of correspondence and conformance link recovery. Recovering correspondence and conformance links is done by implementing heuristics for the Comparative Fragment Analysis API, as mentioned in §4.2.2.4. Heuristics are strategies in sense of the Strategy Pattern. They are used for traversing two ASTs and comparing nodes in a non-probabilistic yet not absolutely correct deterministic way. They employ simple string comparison methods like plural removal, that is deletion of the English plural, and letter case normalization for detecting similarities. This heavily relies on conventions implemented in technologies, e.g. JAXB, where Java field identifiers are used as XML element names, if not declared otherwise. In case, a Java annotation provides an explicit name, it is used with strict equivalence.

Listings 5.10 and 5.11 shows excerpts of the heuristics implemented to uncover related elements among XML and XSD fragments. It uses an abstract base class (listing 5.10) for traversing fragment ASTs and a concrete class for specifying comparison predicates (listing 5.11).

Listing 5.10 Excerpt of the BaseXmlXsdSimilarityHeuristic class

```
public abstract class BaseXmlXsdSimilarityHeuristic implements IFragmentAnalyzerHeuristic {

    protected abstract boolean areSimilar(XmlElementFragment xmlElementFragment, XmlElementFragment
         xsdElementFragment);
    protected abstract boolean areSimilar(XmlAttributeFragment xmlAttributeFragment, XmlElementFragment
         xsdElementFragment);

    ...

    private void addSimilarities(IBinaryRelation<IFragment> similarities, XmlElementFragment
         xmlElementFragment, XmlElementFragment xsdElementFragment) {
        if (areSimilar(xmlElementFragment, xsdElementFragment)) {
            similarities.add(xmlElementFragment, xsdElementFragment);
            for (XmlAttributeFragment xmlAttributeFragment : xmlElementFragment.getXmlAttributeFragments())
                 {
                addSimilarities(similarities, xmlAttributeFragment,
                     xsdElementFragment.getXmlElementFragments());
            }
        }
        for(XmlElementFragment xsdElementFragment1 : xsdElementFragment.getXmlElementFragments()) {
            addSimilarities(similarities, xmlElementFragment, xsdElementFragment1);
        }
        for(XmlElementFragment xmlElementFragment1 : xmlElementFragment.getXmlElementFragments()) {
            addSimilarities(similarities, xmlElementFragment1, xsdElementFragment);
        }
        addSimilarities(similarities, xmlElementFragment.getXmlElementFragments(),
             xsdElementFragment.getXmlElementFragments());
    }

    ...
}
```

Listing 5.11 The XmlXsdSimilarityHeuristic class

```
public class XmlXsdSimilarityHeuristic extends BaseXmlXsdSimilarityHeuristic {
    @Override
    protected boolean areSimilar(XmlElementFragment xmlElementFragment, XmlElementFragment
         xsdElementFragment) {
        return (XmlFragmentUtils.isXsElementTag(xsdElementFragment)
                || XmlFragmentUtils.isXsComplexTypeTag(xsdElementFragment))
                && XmlFragmentUtils.hasAttribute(xsdElementFragment, "name", xmlElementFragment.getName());
    }

    @Override
    protected boolean areSimilar(XmlAttributeFragment xmlAttributeFragment, XmlElementFragment
         xsdElementFragment) {
        return XmlFragmentUtils.isXsAttributeTag(xsdElementFragment)
                && XmlFragmentUtils.hasAttribute(xsdElementFragment, "name",
                      xmlAttributeFragment.getName());
    }
}
```

Figure 5.3 shows an UML class diagram depicting all implemented heuristics. All heuristics follow the same implementation scheme, exemplified with listings 5.10 and 5.11. There is an abstract base class handling traversal and concrete classes handling comparison. Latter classes are further distinguished by the comparison tactic employed by them. They either implement name based similarity predicates or their implementation is based on annotation meta-data. The implemented heuristics detect:

- name based similarities between XML and XSD artifacts, e.g. conformance links between XML elements and XSD complex types or conformance links between XML attributes and XSD attribute definitions;

- name or annotation based similarities between Java and XML or XSD artifacts generated by JAXB, e.g. correspondence links between Java fields and XML attributes or XSD attribute definitions;

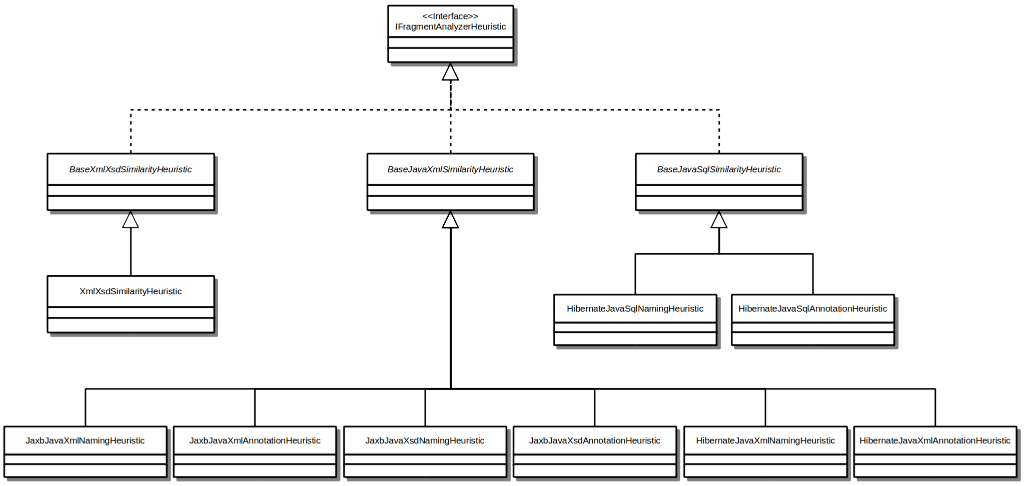

<<Interface>>
IFragmentAnalyzerHeuristic
BaseXmlXsdSimilarityHeuristic
BaseJavaXmlSimilarityHeuristic
BaseJavaSqlSimilarityHeuristic
XmlXsdSimilarityHeuristic
HibernateJavaSqlNamingHeuristic
HibernateJavaSqlAnnotationHeuristic
JaxbJavaXmlNamingHeuristic
JaxbJavaXmlAnnotationHeuristic
JaxbJavaXsdNamingHeuristic
JaxbJavaXsdAnnotationHeuristic
HibernateJavaXmlNamingHeuristic
HibernateJavaXmlAnnotationHeuristic

Figure 5.3 Implemented Heuristics

- name based similarities between Java and Hibernate mapping artifact, e.g. correspondence links between Java fields and Hibernate property mapping definitions;
- name or annotation based similarities between Java and SQL/DDL artifacts generated by Hibernate, e.g. correspondence links between Java fields and SQL column definitions.

Chapter 6

Mini Case-Study

This chapter provides a mini case-study evaluating the developed recovery system for this thesis. §6.1 outlines the corpus used for evaluation. §6.3 describes metrics used to evaluate the system. §6.4 discusses results of the case-study.

6.1 Corpus

The example corpus used to evaluate the recovery system for this thesis consists of artifacts implementing a fictional Human Resource Management System (HRMS) within an O/R/X-Mapping scenario using Java technologies. The model is provided by the 101wiki[1], where it is used for contributions. It is implemented using plain Java and then mapped to plain XML/XSD with JAXB and to SQL/-DDL statements using Hibernate mapping files and/or annotations.

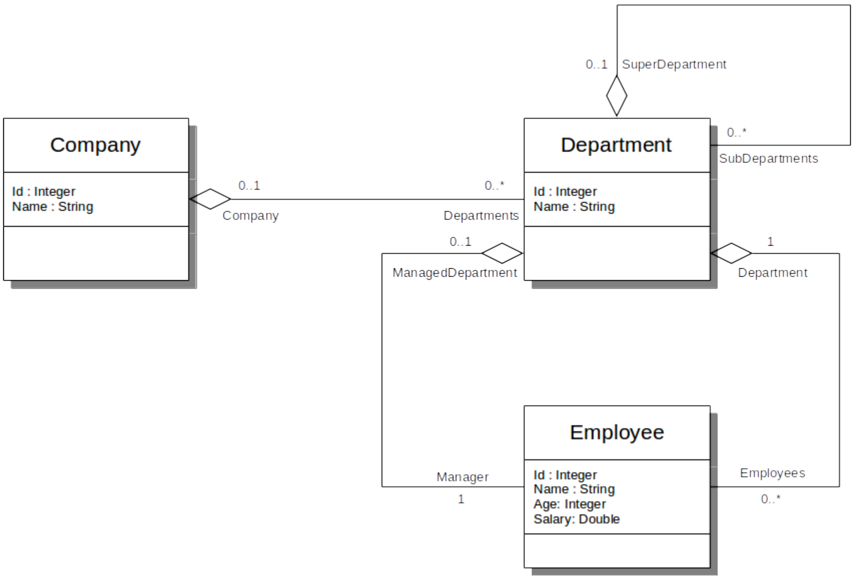
6.1.1 The 101HRMS Model

101wiki Human Resource Management System (101HRMS)[2] provides a simple model of a company with many departments and employees. Figure 6.1 shows an UML class diagram of a variant of this model.

The 101HRMS model consists of companies attributed with a name. Each company accumulates departments. Each department is also attributed with a

[1] `https://101wiki.softlang.org/` (retrieved 12th November, 2017)

[2] `https://101wiki.softlang.org/101:@system` (retrieved 12th November, 2017)



This UML class diagram depicts the model of the 101HRMS. It consists of simple companies with nested departments and employees mapped to the latter.

Figure 6.1 The 101 Human Resource Management System Model

name, aggregates employees and has one employee acting as manager. Departments can further be refined into sub-departments. Each employee is attributed with a name, an age and a salary. Each entity is also attributed with an ID.

6.1.2 Linguistic Domains of the Example Corpus

The example corpus used to evaluate the recovery system contains artifacts implementing the 101HRMS model generated or used by Java technologies for O/R/X-Mapping, i.e. a Java model is mapped to plain XML/XSD with JAXB, to a Hibernate mapping file and to SQL/Data Definition Language (DDL) statements. Figure 6.2 shows a schematic illustration of the linguistic domains involved:

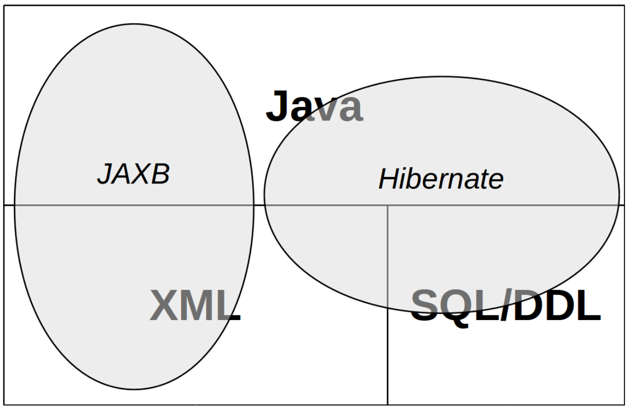
Java The language and technology used to implement the 101HRMS model.

XML The language used to serialize the 101HRMS model.

SQL/DDL The language used to persist the 101HRMS model.

JAXB The technology used to implement O/X-Mapping of the 101HRMS model.

Hibernate The technology used to implement O/R-Mapping of the 101HRMS model.



This schematic illustration depicts the interrelation among linguistic domains of example corpus used. It depicts languages and technologies for O/R/X-Mapping with Java.

Figure 6.2 Example Corpus Domains: Java O/R/X

The languages (Java, XML & SQL) in Figure 6.2 are displayed as disjoint square sets. Technologies (JAXB & Hibernate) are displayed as oval sets intersecting languages. This is due to their linguistic nature, e.g. JAXB produces specific Java- and XML-Code which does not necessarily intersect with code produced by other technologies. Hibernate intersects all three languages. It uses XML files or Java-Annotations for describing O/R-Mapping of a data-model and generates SQL artifacts according to that mapping. In this sense, technologies create technology-specific subsets of a languages.

6.2 Setup

The implemented recovery system is evaluated with the megamodel shown in listing 6.1.

Listing 6.1 Megamodel Setup

```
companyJavaFile : File
companyJavaFile = 'workspace:/org.softlang.megal.plugins/input/Company.java'

companyHbmFile : File
companyHbmFile = 'workspace:/org.softlang.megal.plugins/input/Company.hbm.xml'

```

```
departmentJavaFile : File
departmentJavaFile = 'workspace:/org.softlang.megal.plugins/input/Department.java'

departmentHbmFile : File
departmentHbmFile = 'workspace:/org.softlang.megal.plugins/input/Department.hbm.xml'

employeeJavaFile : File
employeeJavaFile = 'workspace:/org.softlang.megal.plugins/input/Employee.java'

employeeHbmFile : File
employeeHbmFile = 'workspace:/org.softlang.megal.plugins/input/Employee.hbm.xml'

companiesXmlFile: File
companiesXmlFile = 'workspace:/org.softlang.megal.plugins/input/companies.xml'

companiesXsdFile: File
companiesXsdFile = 'workspace:/org.softlang.megal.plugins/input/companies.xsd'

comaniesSqlFile : File
comaniesSqlFile = 'workspace:/org.softlang.megal.plugins/input/companies.ddl.sql'

companyJavaFile correspondsTo companiesXsdFile
companyJavaFile correspondsTo comaniesSqlFile
companyJavaFile correspondsTo companyHbmFile

departmentJavaFile correspondsTo companiesXsdFile
departmentJavaFile correspondsTo comaniesSqlFile
departmentJavaFile correspondsTo departmentHbmFile

employeeJavaFile correspondsTo companiesXsdFile
employeeJavaFile correspondsTo comaniesSqlFile
employeeJavaFile correspondsTo employeeHbmFile

companiesXmlFile conformsTo companiesXsdFile
```

We declare nine artifacts to be files and bind them to locations on the local file system. For each class of the 101HRMS model, i.e. company, department and employee, we declare a pair of entities comprising a Java class file and a Hibernate mapping file. For the 101HRMS model at whole, artifacts for its XML, XSD and SQL/DDL representations are declared. Following this, we declare the expected correspondence and conformance relations among the files.

6.3 Metrics

We evaluate the implemented recovery system regarding the requirements stated in §4.1, i.e. requirements 1, 2, 3 and 4 for recovery of parthood, fragments, correspondence links and conformance links.

For this, we measure the quantity of all entities of concern grouped by type, i.e. artifacts, files and fragments:

$$\begin{aligned}\#\mathsf{Artifact} &:= |\{x : \mathsf{Artifact}(x)\}| \\ \#\mathsf{File} &:= |\{x : \mathsf{File}(x)\}| \\ \#\mathsf{Fragment} &:= |\{x : \mathsf{Fragment}(x)\}|\end{aligned}$$

We also measure the overall amount of all relevant links grouped by type, i.e. partOf, fragmentOf, correspondsTo and conformsTo:

$$\begin{aligned}\#\mathsf{partOf} &:= |\{(x, y) : \mathsf{partOf}(x, y)\}| \\ \#\mathsf{fragmentOf} &:= |\{(x, y) : \mathsf{fragmentOf}(x, y)\}| \\ \#\mathsf{correspondsTo} &:= |\{(x, y) : \mathsf{correspondsTo}(x, y)\}| \\ \#\mathsf{conformsTo} &:= |\{(x, y) : \mathsf{conformsTo}(x, y)\}|\end{aligned}$$

For a more differentiated look on requirements 1 and 2, we measure the number of parts and fragments of a specific entity:

$$\begin{aligned}\#\mathsf{partOf}(x) &:= |\{y : \mathsf{partOf}(y, x)\}| \\ \#\mathsf{fragmentOf}(x) &:= |\{y : \mathsf{fragmentOf}(y, x)\}|\end{aligned}$$

For requirements 3 and 4, i.e. correspondence and conformance recovery, we devise a similar measurement. However, we do not count the number of entities corresponding or conforming to a given other entity. We rather count entities corresponding or conforming to fragments of the given entity.

$$\begin{aligned}\#\mathsf{correspondsTo}(y) &:= |\{x : \mathsf{correspondsTo}(x, y') \wedge \mathsf{fragmentOf}(y', y)\}| \\ \#\mathsf{conformsTo}(y) &:= |\{x : \mathsf{conformsTo}(x, y') \wedge \mathsf{fragmentOf}(y', y)\}|\end{aligned}$$

Eventually, we are interested in the number of unique elements per relationship, distinguishing left-hand side (LHS) from right-hand side (RHS) entities:

$$\begin{aligned}\#LHS(R) &:= |\{x : R(x, y)\}| \\ \#RHS(R) &:= |\{y : R(x, y)\}|\end{aligned}$$

6.4 Results

An evaluation of the setup described in §6.2 with the metrics of §6.3 provides the following results. Table 6.1 shows the number of all artifacts, files and fragments in the megamodel after execution. It indicates that files and fragments are disjoint as proposed by axiom 2 in §2.1. All files and all fragments are artifacts, so their numbers sum up to the number of artifacts: #Artifact = #File + #Fragment. The number of files matches the given setup (see §6.2); the remaining artifacts are fragments.

#Artifact	#File	#Fragment
483	9	474

Table 6.1 Number of recovered artifacts, files and fragments

Table 6.2 shows the number of recovered trace links for parthood, fragment, correspondence and conformance relations. The numbers for partOf- and fragmentOf- relationships do not match. This can be explained with the usage of partOf to configure plug-ins by MegaL/Xtext [14]. Moreover, the numbers for parthood and fragment links are relatively large compared to the overall number of artifacts in table 6.1. This may be due to the simplicity of the metric. It simply counts all unique pairs. Parthood and fragment links are transitive, which leads to this accumulation.

#partOf	#fragmentOf	#correspondsTo	#conformsTo
1847	1837	106	81

Table 6.2 Number of recovered parthood, fragment, correspondence and conformance links

On the other hand, the number of recovered correspondence and conformance links are relatively small compared to the number of Parthood and fragment links. This can be explained by the nature of the recovery method in conjunction with the setup. Java classes and the majority of classical object-oriented programming languages group behavior and structure. A class may contain field- and method-fragments, which are also recovered. However, only the former may be linked through correspondence and conformance within the given setup.

Table 6.3 shows the number of recovered parts and fragments per file. The overall equivalence $\#\mathsf{partOf}(x) = \#\mathsf{fragmentOf}(x)$ for all files indicates that all parts of each file are indeed proper parts (see §2.1.2). Furthermore it indicates in conjunction with table 6.1 that all recovered fragments are properly linked to the files their originated from, since sum of all recovered fragments per file matches the number of all recovered fragments.

x	$\#\mathsf{partOf}(x)$	$\#\mathsf{fragmentOf}(x)$
companyJavaFile	9	9
companyHbmFile	51	51
departmentJavaFile	20	20
departmentHbmFile	112	112
employeeJavaFile	15	15
employeeHbmFile	76	76
companiesXmlFile	89	89
companiesXsdFile	85	85
comaniesSqlFile	17	17
Sum:	474	474

Table 6.3 Number of parts and fragments per file

Table 6.4 shows the number of entities corresponding of conforming to fragments of a file. For instance the twelve entities corresponding to a fragment of companyJavaFile are shown in listing 6.2, which shows that all Java fields are

x	$\#\mathsf{correspondsTo}(x)$	$\#\mathsf{conformsTo}(x)$
companyJavaFile	12	0
companyHbmFile	4	0
departmentJavaFile	15	0
departmentHbmFile	5	0
employeeJavaFile	17	0
employeeHbmFile	5	0
companiesXmlFile	0	0
companiesXsdFile	16	67
comaniesSqlFile	11	0

Table 6.4 Number of entities corresponding or conforming to fragments of a file

linked to corresponding XSD attribute declarations, Hibernate mapping declarations or SQL/DDL column declarations, except for fields implying a foreign key relationship.

Listing 6.2 Recovered Correspondences

```
comaniesSqlFile.F3$Company correspondsTo companyJavaFile.Company
comaniesSqlFile.F3$Company.F0$name correspondsTo companyJavaFile.Company.F5$name
comaniesSqlFile.F3$Company.F1$id correspondsTo companyJavaFile.Company.F6$id

companiesXsdFile.F0$xs:schema.F2$xs:element correspondsTo companyJavaFile.Company
companiesXsdFile.F0$xs:schema.F5$xs:complexType correspondsTo companyJavaFile.Company
companiesXsdFile.F0$xs:schema.F5$xs:complexType.F2$xs:attribute correspondsTo companyJavaFile.Company.F6$id
companiesXsdFile.F0$xs:schema.F5$xs:complexType.F3$xs:attribute correspondsTo
    companyJavaFile.Company.F5$name
companiesXsdFile.F0$xs:schema.F5$xs:complexType.F1$xs:sequence.F0$xs:element correspondsTo
    companyJavaFile.Company.F4$departments

companyHbmFile.F0$hibernate-mapping.F4$class correspondsTo companyJavaFile.Company
companyHbmFile.F0$hibernate-mapping.F4$class.F10$property correspondsTo companyJavaFile.Company.F5$name
companyHbmFile.F0$hibernate-mapping.F4$class.F8$id correspondsTo companyJavaFile.Company.F6$id
companyHbmFile.F0$hibernate-mapping.F4$class.F9$bag correspondsTo companyJavaFile.Company.F4$departments
```

$\#\textsf{correspondsTo}(\text{companiesXmlFile}) = 0$ indicates that correspondence does not occur between model- and instance-level artifacts. This satisfies the definition of correspondence, denoting two artifacts represent the same data (see §2.1.3), which should not occur between different levels of abstraction.

Regarding conformance, the table indicates, that it only occurs towards XSD files, since $\#\textsf{conformsTo}(x)$ is zero for all files except companiesXsdFile. This is in accordance with the definition of conformance, given within the setup XSD is the only specification language for the other artifacts present.

Table 6.5 shows the number of unique entities per relationship. We distinguish left-hand side (LHS) from right-hand side (RHS) entities. The number of corresponding and conforming entities is significantly lower than the overall number of recovered fragments in table 6.1. This may indicate recovery of unnecessary fragments, e.g. Javas method fragments; or it indicates an insufficiency of the implemented correspondence and conformance recovery. However,

R	$\#LHS(R)$	$\#RHS(R)$
partOf	484	121
fragmentOf	474	117
correspondsTo	68	68
conformsTo	68	19

Table 6.5 Number of unique entities per relationship

the number of unique left-hand side entities for conformsTo matches the number of #conformsTo(companiesXsdFile) minus one, the latter being the initial conformsTo-declaration from the setup (see §6.2). This assures that conformance is only recovered towards XSD. Likewise the number of unique left-hand side entities for fragmentOf matches the number of all recovered fragments from table 6.1, the delta to partOf remaining the same for both sides.

Chapter 7

Conclusion

We showed how static program analysis can be used for trace link recovery among source code artifacts with trace semantics derived from an axiomatization of linguistic architectures (see §2). For this, we designed and implemented an exemplary system capable of recovering parthood and fragment traces as well as correspondence and conformance links (see §4 and §5). This system was then evaluated against a minimal corpus facilitating a mini case-study (see §6).

7.1 Future Work

Larger Corpus §6 only evaluates the implemented recovery system against a minimal corpus. In order to provide more reliable results, future work should reevaluate the system against a larger, possibly more heterogeneous corpus.

Instance-Level Trace Link Recovery The presented system only recovers instance-level trace links in form of XML and XSD conformance links. Future work should try to also recover instance-level correspondence links. However, static program analysis and trace recovery as defined in §2.2 may not be suitable for this task, Since instances are not necessarily available as persistent artifacts for all languages used, an application of *trace capture* [13] may be advised.

Foreign Key Trace Link Recovery Correspondence link recovery for SQL/-DDL as described in §5 does not consider foreign key relations. Hence, recovery of correspondence links among SQL and Java artifacts is incomplete. Future work should implement this missing feature.

Visualization The presented system only recovers trace links and persists them using the MegaL/Xtext environment. Future work may include development of suitable visualization for trace links in the context of linguistic architectures.

Program Comprehension Traceability with semantics of linguistic architectures exemplified in §2.1 may be used to support program comprehension. Future work should examine prospects in that direction, e.g. regarding bottom-up comprehension models [28].

Glossary

101HRMS 101wiki[1] Human Resource Management System[2]. The model used by the 101wiki for its contributions. 46–49, see also HRMS

Abstract Factory Pattern A creational GoF (Gang of Four) pattern used in software design to decouple instantiation from usage of objects. Hides the concrete nature of created instances. 27, 28, 31

ANTLR Another Tool For Language Recognition (see [27]). 14, 18, 19, 26–29, 36, 39

API Application Programming Interface. 5, 16, 25–27, 29–32, 36, 43, 58, 61

artifact An object created during a software development process for a certain purpose. The term artifact usually refers to a digital document or a well-formed part of it. 3–5, 10–12, 16, 20–22, 24, 37, 44, 45, 49–51, 53, 55, 56, 58, 61, B

AST Abstract Syntax Tree: A tree data structure representing the abstract syntax of a parsed text. This tree omits syntactic features like parentheses for grouping or semicolons for sequencing. 4, 5, 18, 20, 25–27, 29–32, 36, 37, 39, 41, 43

Builder Pattern A creational GoF pattern used in software design to prevent constructor parameters from piling up. 30, 31

[1] `https://101wiki.softlang.org/` (retrieved 12th November, 2017)

[2] `https://101wiki.softlang.org/101:@system` (retrieved 12th November, 2017)

conformance The relation between two artifacts, denoting one defines the other like a metamodel defines a model (see §2.1, axiom 4). 3, 5, 7, 9, 11, 12, 32, 35, 43, 44, 49–51, 53–55

correspondence The relation between two artifacts, denoting they represent the same data or encode the same information (see §2.1, axiom 3). Usually both artifacts only differ syntactically. 3, 5, 7, 9–11, 24, 32, 35, 43–45, 49–51, 53, 55, 56

CST Concrete Syntax Tree: A tree data structure representing the concrete syntax of a parsed text.. 25, 26

DDL Data Definition Language. Language or subset of a language used to describe structure and content of data. 47

DFS The algorithmic concept of traversing a tree or graph data structure 'top-down' until reaching the end of a path before backtracking and traversing another path. 19, 25, 29, 42, 43

DTO Data Transfer Object. Objects with no relevant (business) logic of their own. Their sole purpose is to carry data between layers of a software system. 26

dynamic program analysis The automated analysis of a program's behavior and side-effects while executing it. It is the opposite of static program analysis. 60

ER Model Entity-Relationship Model. 3, 15, 60

Facade Pattern A structural GoF pattern used in software design to simplify the usage of complex systems or APIs. It provides single access point for such system. Such access points are called facades. 33

fragment A syntactically well-formed piece of a possibly larger text. 4, 5, 9, 10, 24, 35–41, 43, 49–55

GoF Gang of Four. A group of authors (Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides) publishing on the subject of object-oriented software design. The term may also refer to design patterns described in their book *Design Patterns: Elements of Reusable Object-Oriented Software* [10]. 57–61

Hibernate The Hibernate ORM Framework. 5, 14, 17, 18, 32, 33, 38, 45–49, 53

HRMS Human Resource Management System. 46

HTTP Hypertext Transfer Protocol. 1

IDE Integrated Development Environment. 15, 22

Iterator Pattern A behavioral GoF pattern used in software design to traverse/iterate elements of a container data structure. 43

Java The Java Programming Language and Platform. 2, 3, 5, 7, 9, 16–18, 21, 28, 32, 33, 35–41, 43–49, 51–53, 56

JAXB Java Architecture for XML Binding. 2, 5, 14, 16, 17, 32, 33, 38, 43, 44, 46–48

JPA Java Persistence API. 18

JSON JavaScript Object Notation. 10

language Umbrella term for natural, formal and programming languages. In context of this thesis, the term is usually refers to the latter two, unless noted otherwise. 1, 3, 10, 12, 53, 55, 59, 61, B

linguistic architecture The architecture or model of a software system concerned with its outline regarding languages and technologies used by it or used to build it (see [9], [20] and [15]). 3, 7, 10, 15, 55, 56, B

manifestation . 3, 10, see representation

MegaL The megamodeling language developed by the Softlang Team at the University of Koblenz-Landau for descriptively and prescriptively modeling linguistic architectures of software systems. 14–16, 33, 41, 59

MegaL/Xtext The Xtext implementation and eclipse IDE integration of MegaL. 14–16, 24, 25, 33, 34, 51, 56, 62

megamodel A model of models. Megamodels describe relations among different kinds of models, e.g. the relations between models and their metamodels or models and their instances. 3, 15, 16, 33, 36, 37, 39–41, 48, 51

mereology The philosophical and logical discipline of studying the constituent parts of things and the relations in between (see [30] and [29]). 7–9, 30

metamodel The syntactic and semantic specification of a model. 3, 58

O/R-Mapping Object-Relational-Mapping. The bidirectional mapping between objects in the sense of object-oriented programming and a relational database system. It usually maps classes to tables and instances to rows of a table. 17, 33, 48

O/R/X-Mapping Object-Relational- and XML-Mapping. 1, 2, 5, 25, 32, 38, 46–48

O/X-Mapping Object-XML-Mapping. The bidirectional mapping between objects in the sense of object-oriented programming and their XML representations. It usually maps classes to XSD and instances to XSD entities. 16, 47

Observer Pattern A behavioral GoF pattern used in software design to propagate state changes from one object to many dependent objects. 19, 29

ontology Domain knowledge captured in form of an ER Model. 3

ORM . 59, see O/R-Mapping

Parse Tree . 4, 18, 19, 25, 28, 29, 36, 37, 39, 41, see CST

parthood The relation between an entity and its constituent parts. 6–10, 12, 24, 30–32, 35, 40–42, 49, 51, 55

representation The syntactic encoding of information or a conceptual model. 3, 10, 60

SQL Structured Query Language. 18, 33, 45, 47, 48, 56

SQL/DDL The DDL subset of SQL. 32, 33, 36, 45, 46, 49, 53, 55

static program analysis The automated analysis of programs without executing them. It is the opposite of dynamic program analysis. 4, 5, 14, 22, 55, 58, B

Strategy Pattern A behavioral GoF pattern used in software design to separate behavior from structure. It allows to encapsulate and reuse behavior as part of the configuration of larger constructs. 31, 43

technology Programs, languages, APIs, etc. used in a software development process. 1, 3, 14, 59, B

trace A trace link or the act of following that link [13] (see §2.2.2). 13, 55, 61

trace link A semantic link between two artifacts [13] (see §2.2.2). 3–5, 51, 55, 56, 61, B

trace link recovery . see trace recovery

trace recovery The action of creating traces among already existing artifacts [13] (see §2.2.2). 3

traceability The potential for traces to be established and used [13] (see §2.2.2). 1, 3, 12, 13, 56, B

UML Unified Modeling Language. 27, 30, 31, 36, 44, 46, 47

Visitor Pattern A behavioral GoF pattern used in software design to separate behavior from structure. Visitors facilitate the extension of behavior without modifying structure. The Visitor Pattern can be used to traverse object graphs. 19, 29

XML Extensible Markup Language. 1–3, 10, 16–18, 21, 32, 33, 36, 40, 41, 43, 44, 46–49, 55, 60

XSD XML Schema Definition. 1–3, 16, 33, 43, 44, 46, 47, 49, 53–55, 60

Addendum

Source codes of this thesis, the implemented recovery system and its integration into MegaL/Xtext is available on GitHub:

`https://github.com/maxmeffert/BScThesis`

Source code of the original implementation of MegaL/Xtext is also available on GitHub:

`https://github.com/avaranovich/megal-xtext`

Bibliography

[1] Elian Angius and René Witte. "OpenTrace: An Open Source Workbench for Automatic Software Traceability Link Recovery". In: *WCRE*. IEEE Computer Society, 2012, pp. 507–508.

[2] Giuliano Antoniol et al. "Recovering Code to Documentation Links in OO Systems". In: *WCRE*. IEEE Computer Society, 1999, pp. 136–144.

[3] Giuliano Antoniol et al. "Tracing Object-Oriented Code into Functional Requirements". In: *IWPC*. IEEE Computer Society, 2000, pp. 79–86.

[4] Anya Helene Bagge and Vadim Zaytsev. "Languages, Models and Megamodels". In: *SATToSE*. Vol. 1354. CEUR Workshop Proceedings. CEUR-WS.org, 2014, pp. 132–143.

[5] Gabriele Bavota et al. "TraceME: Traceability Management in Eclipse". In: *ICSM*. IEEE Computer Society, 2012, pp. 642–645.

[6] Oracle Corporation. *Java Architecture for XML Binding*. `http://www.oracle.com/technetwork/articles/javase/index-140168.html`. Retrieved 11th December, 2017.

[7] Oracle Corporation. *Steps in the JAXB Binding Process*. Retrieved 11th December, 2017. URL: `https://docs.oracle.com/javase/tutorial/figures/jaxb/jaxb-dataBindingProcess.gif`.

[8] Oracle Corporation. *The Java Tutorials: Java Architecture for XML Binding*. `https://docs.oracle.com/javase/tutorial/jaxb/TOC.html`. Retrieved 11th December, 2017.

[9] Jean-Marie Favre, Ralf Lämmel, and Andrei Varanovich. "Modeling the Linguistic Architecture of Software Products". In: *MoDELS*. Vol. 7590. Lecture Notes in Computer Science. Springer, 2012, pp. 151–167.

[10] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[11] Orlena Gotel and Anthony Finkelstein. "An analysis of the requirements traceability problem". In: *ICRE*. IEEE Computer Society, 1994, pp. 94–101.

[12] Orlena Gotel et al. "Appendix A: Glossary of Traceability Terms (v1.0)". In: *Software and Systems Traceability.* Springer, 2012, pp. 413–424.

[13] Orlena Gotel et al. "Traceability Fundamentals". In: *Software and Systems Traceability*. Springer, 2012, pp. 3–22.

[14] Lukas Härtel. "Linguistic architecture on the workbench". Bachelor Thesis. University of Koblenz-Landau, Oct. 2015.

[15] Marcel Heinz, Ralf Lämmel, and Andrei Varanovich. "Axioms of Linguistic Architecture". In: *MODELSWARD*. SciTePress, 2017, pp. 478–486.

[16] Red Hat Inc. *Hibernate ORM*. `http://hibernate.org/orm/l`. Retrieved 11th December, 2017.

[17] Huzefa H. Kagdi, Jonathan I. Maletic, and Bonita Sharif. "Mining Software Repositories for Traceability Links". In: *ICPC*. IEEE Computer Society, 2007, pp. 145–154.

[18] Sam Klock et al. "Traceclipse: an eclipse plug-in for traceability link recovery and management". In: *EFSE@ICSE*. ACM, 2011, pp. 24–30.

[19] Ralf Lämmel. "Coupled software transformations revisited". In: *SLE*. ACM, 2016, pp. 239–252.

[20] Ralf Lämmel and Andrei Varanovich. "Interpretation of Linguistic Architecture". In: *ECMFA*. Vol. 8569. Lecture Notes in Computer Science. Springer, 2014, pp. 67–82.

[21] Andrea De Lucia et al. "Information Retrieval Methods for Automated Traceability Recovery". In: *Software and Systems Traceability*. Springer, 2012, pp. 71–98.

[22] Andrea De Lucia et al. "Recovering traceability links in software artifact management systems using information retrieval methods". In: *ACM Trans. Softw. Eng. Methodol.* 16.4 (2007), p. 13.

[23] Collin McMillan, Denys Poshyvanyk, and Meghan Revelle. "Combining textual and structural analysis of software artifacts for traceability link recovery". In: *TEFSE@ICSE*. IEEE Computer Society, 2009, pp. 41–48.

[24] Richard Freeman Paige et al. "Building Model-Driven Engineering Traceability Classifications". In: (Jan. 2010).

[25] Terence Parr. *ANTLR*. `http://www.antlr.org/`. Retrieved 12th December, 2017.

[26] Terence Parr. *Getting Started with ANTLR v4*. `https://github.com/antlr/antlr4/blob/master/doc/getting-started.md`. Retrieved 12th December, 2017.

[27] Terence Parr. *The Definitive ANTLR 4 Reference*. 2nd. Pragmatic Bookshelf, 2013. ISBN: 1934356999, 9781934356999.

[28] Janet Siegmund. "Program Comprehension: Past, Present, and Future". In: *FOSE@SANER*. IEEE Computer Society, 2016, pp. 13–20.

[29] Achille C. Varzi. "Mereology". In: *The Stanford encyclopedia of philosophy*. Ed. by Edward N. Zalta. Winter 2016 ed. 2016. URL: `https://plato.stanford.edu/archives/win2016/entries/mereology/`.

[30] Achille C. Varzi. "Parts, Wholes, and Part-Whole Relations: The Prospects of Mereotopology". In: *Data Knowl. Eng.* 20.3 (1996), pp. 259–286.

[31] Stefan Winkler and Jens von Pilgrim. “A survey of traceability in requirements engineering and model-driven development”. In: *Software and System Modeling* 9.4 (2010), pp. 529–565.

[32] Andrea Zisman. “Using Rules for Traceability Creation”. In: *Software and Systems Traceability*. Springer, 2012, pp. 147–170.