Julius Köpke

Declarative Semantic Annotations for XML Document Transformations and their Maintenance

Dissertation

zur Erlangung des akademischen Grades
Doktor der Technischen Wissenschaften

Alpen-Adria-Universität Klagenfurt
Fakultät für Technische Wissenschaften

1. Begutachter: O.Univ.-Prof. DI Dr. Johann Eder
Universität Klagenfurt / Institut für Informatik Systeme

2. Begutachter: Prof. Dr. Michele Missikoff
IASI-CNR, Rome

March/2012

Declaration of honour

I hereby confirm on my honour that I personally prepared the present academic work and carried out myself the activities directly involved with it. I also confirm that I have used no resources other than those declared. All formulations and concepts adopted literally or in their essential content from printed, unprinted or Internet sources have been cited according to the rules for academic work and identified by means of footnotes or other precise indications of source. The support provided during the work, including significant assistance from my supervisor has been indicated in full. The academic work has not been submitted to any other examination authority. The work is submitted in printed and electronic form. I confirm that the content of the digital version is completely identical to that of the printed version. I am aware that a false declaration will have legal consequences.

Julius Köpke Klagenfurt, 2012 March 26

Acknowledgements

I'm deeply grateful to my mentor Prof. Johann Eder. His valuable comments, fruitful discussions and the process of working together on publications had a strong impact on my research skills in general and on this thesis in particular. In addition I want to thank him for providing a good working environment that allowed me to focus on research.
I also want to thank Prof. Michele Missikoff for peer-reviewing this thesis.

The next group of persons I want to thank are my colleagues from our institute. I especially want to thank my office-mate Nico Kerschbaumer for many good research discussions and for always being helpful in many different occasions - especially concerning Latex problems.

Another group of persons that were involved in this research are the master students Marcin Szymczak and Hannes Hannig who partially worked on the implementation of my research. My special thanks go to Marcin who worked with strong commitment on the implementation of the matching engine.

Working on a dissertation has no boundaries with regard to time and location. This requires patience from the persons in one's environment. I want to thank my family and friends for their support and patience. This holds especially for Sabrina. To her I dedicate this thesis.

Abstract

Semantic annotations of XML-Schemas allow the interpretation of the schema elements with the help of a reference ontology. This can be a driver for the integration of heterogenous applications, where the exchanged messages needs to be transformed to the required target format. Since an ontology conceptualizes some real-world domain and the real world constantly evolves also the reference ontology needs to evolve over time. This has consequences for the semantic annotations that need to be maintained in order to comply with the new version of the reference ontology. In addition such an evolving reference ontology gets an additional function: It does not only express the domain at some specific point in time - it can also express the evolution of the domain. This information can be used to (semi-) automatically maintain the annotations and to detect changes that have consequences for the interpretation of instance data. This may require to change instance data in order to allow a correct interpretation with the latest ontology version. In this thesis we will present a purely declarative annotation method for XML-Schemas, matching and mapping methods for schemas that are annotated with the proposed annotations, methods for the representation of ontology changes, methods for the maintenance of annotations with regard to structure and logics and finally the detection of schema elements, where the interpretation of the data has potentially changed.

Contents

1 Introduction **1**
1.1 Contents of the Thesis . . . 3
1.1.1 Semantic Annotation . . . 3
1.1.2 Schema Mapping . . . 3
1.1.3 Change Representation . . . 5
1.1.4 Annotation Maintenance . . . 6
1.1.5 Detection of Semantic Changes . . . 6
1.2 Preliminaries . . . 6
1.2.1 Interoperability . . . 7
1.2.2 XML-Schema . . . 7
1.2.3 Ontologies . . . 7
1.2.4 Ontology Formalisms and Languages . . . 8

2 Semantic Annotation **11**
2.1 Motivating Example . . . 12
2.2 Annotation Method . . . 14
2.2.1 Formal Definition of the Annotation Method . . . 15
2.2.2 Reuse of Global Types or Elements . . . 16
2.3 Transformation of Annotation Paths to Ontology Concepts . . . 17
2.4 Validation of Annotations . . . 18
2.5 Related Work . . . 19
2.6 Conclusion . . . 20

3 XML-Schema Matching and Mapping **21**
3.1 Introduction . . . 21
3.1.1 Schema Matching . . . 22
3.1.2 Complex Matches . . . 24
3.1.3 Matching Approaches for XML-Schema . . . 26
3.2 Annotation based XML-Schema Matching . . . 27
3.2.1 Schema Mapping Model . . . 27
3.2.2 Semantic Relations between Annotations for Simple Matches . . . 28

3.2.3 Semantic Relations between Annotations for Complex Matches 30
3.2.4 Mapping Workflow 36
3.3 Mismatch Resolution 38
3.3.1 Lossless Mismatches 38
3.3.2 Lossy Mismatches 42
3.3.3 Discussion 43
3.4 Proof of Concept Implementation 44
3.4.1 Annotation Matching Phase 44
3.4.2 Node Mapping Phase 46
3.4.3 Output Generation 47
3.5 Performance Evaluation 47
3.5.1 Lifting/Lowering Implementation 48
3.5.2 Evaluation Setting 49
3.5.3 Experimental Results 50
3.6 Conclusion 53

4 Change Representation **55**
4.1 Change Representation Requirements 55
4.2 Differences between OWL and Frame-Based Ontologies 56
4.3 Ontology Changes 58
4.4 Survey 59
4.4.1 Ontology Evolution Management 60
4.4.2 Ontology Comparison Approaches 61
4.4.3 Change-Tracing Approaches 64
4.4.4 Change Modeling 66
4.4.5 Ontology Evolution Systems 68
4.4.6 Ontology Mapping and Multi-Version Reasoning 69
4.4.7 Discussion of the Approaches 70
4.5 Change Representation Approach 73
4.5.1 Meta Ontology 74
4.5.2 Change Ontology 75
4.5.3 Implementation 78
4.6 Conclusion 80

5 Structural Maintenance of Annotations **83**
5.1 Annotation Change Operations 83
5.1.1 Problem Definition 83
5.1.2 Local Changes 84
5.1.3 Global Changes 84
5.1.4 Change Transactions 84
5.2 Structural Invalidation of Annotation Paths 85

5.3 Evolution Strategies for Structurally Invalid Paths . . . 87
5.3.1 Atomic Evolution Strategies for Missing-Reference-Invalidations . . . 87
5.3.2 Composite Evolution Strategies for Missing-Reference-Invalidations . . . 89
5.3.3 Evolution Strategies for Wrong-Type Invalidations . . . 92
5.4 Annotation Maintenance using Mapping Composition . . . 94
5.5 Conclusion . . . 95

6 Logical Invalidation of Annotations **97**
6.1 Annotation Method . . . 98
6.2 Logical Invalidation of Annotation Paths . . . 99
6.2.1 Invalidation of Simple Concept Annotations . . . 100
6.2.2 Invalidation of Simple Datatype Annotations: . . . 100
6.2.3 Invalidation of 3-Step Concept Annotations . . . 101
6.3 Invalidation of General Annotations . . . 102
6.3.1 Invalidation of General Annotation due to Local Invalidations . . . 103
6.3.2 Direct-Triple-Disjointness . . . 104
6.3.3 Non-Local Invalidations . . . 105
6.3.4 An Algorithm for the Detection of a Minimal Invalid Sub-Path . . . 110
6.4 Implementation Considerations . . . 112
6.5 Additional Justifications of Invalidations in Annotation Paths . . . 113
6.5.1 General Justifications . . . 113
6.5.2 Justifications after Ontology Evolution . . . 115
6.6 Related Work . . . 116
6.7 Conclusion . . . 117

7 Detection of Semantic Changes **119**
7.1 Semantic Changes and their Automatic Detection . . . 119
7.2 Requirements for Explicit Dependency-Definitions . . . 122
7.3 Definition of Change-Dependencies . . . 123
7.3.1 Integrity Constraints on Annotation Path . . . 124
7.3.2 Integrity Constraints on Dependency Definition Path . . . 125
7.4 Detection of Semantic Changes . . . 125
7.5 Proof of Concept Implementation . . . 130
7.6 Related Work . . . 130
7.7 Conclusion . . . 131

8 Case Study **133**
8.1 The Setting . . . 133
8.2 Example Ontology . . . 134
8.3 Example Annotations . . . 136
8.4 Mapping Generation . . . 137
8.5 Ontology Changes . . . 137

8.6 Structural and Logical Annotation Maintenance 138
8.7 Detection of Semantic Changes 141
8.8 Conclusion 142

9 Conclusion **143**

10 Appendix **147**

List of Figures

1.1 XML document transformations using the lifting/lowering approach 4
1.2 XML document transformations using annotation based schema-mapping . . . 5

2.1 Example reference ontology 12
2.2 Example XML-Schema with standard model-references 13
2.3 Example XML-Schema with the proposed annotation method 14

3.1 Match processing in COMA [21] 24
3.2 A Meta-Model of the proposed XML-Schema mapping approach 28
3.3 Finding complex matches with annotated transformation templates 32
3.4 Schema mapping workflow 37
3.5 Example reference ontology 39
3.6 Example mapping 40
3.7 Phases of the proof of concept implementation 44
3.8 Screenshot of the semantic matching phase of the prototype 45
3.9 Generated output opened in Altova MapForce 48
3.10 Transformation duration in seconds for n documents containing 1 item 51
3.11 Transformation duration in seconds for n documents containing 10 items 52
3.12 Transformation duration in seconds of n documents containing 100 items 53

4.1 5-phase ontology evolution approach of [80] 65
4.2 The main classes of the OWL2 change ontology [76] 67
4.3 Proposed change representation approach 74
4.4 Ontology meta-model 75
4.5 Class hierarchy of the change ontology 76

5.1 Example ontology for evolution strategies for structural invalidations 89
5.2 Annotation maintenance using mapping-composition 95

6.1 Example of direct-triple-disjointness 104
6.2 The black-box MUPS algorithm from [45] 114

7.1 Example ontology 120

7.2 Meta-model of the change-dependency definitions 125

8.1 Building-blocks for semantic annotations in an evolving environment 134
8.2 Example car ontology first version 135
8.3 Screen-shot of the mapping-prototype 137
8.4 Automatically generated mapping 138
8.5 Example car ontology second version 139
8.6 Instances of the change ontology 140

1 Source of mismatch example source schema page 1 148
2 Source of mismatch example source schema page 2 149
3 Source of mismatch example target schema 150

Listings

2.1 Representation of a concept annotation path in OWL 17
2.2 Representation of a datatype annotation path in OWL 18
4.1 Example SPARQL query for the detection of added subconcepts 80
5.1 Structural validation of an annotation path 86
5.2 Algorithm for the generation of candidate replacements for invalid concept-steps 91
5.3 Definition of getReplacementConcept() 91
6.1 Representation of an annotation path in OWL 98
6.2 Example ontology for direct-triple-disjointness invalidations 104
6.3 A non local invalidation 105
6.4 Example ontology for an indirect restriction chain 107
6.5 Example ontology for a non local invalidation with a non-inverse-chain 107
6.6 Example ontology for a non local invalidation with an indirect non-inverse-chain 108
6.7 Example ontology for a MIS that is caused by transitive properties 108
6.8 Example ontology for a non local invalidation by a property chain 109
6.9 Example ontology for a non local invalidations by a mixed chain 110
6.10 An algorithm for the detection of the minimal invalid sub-path 111
8.1 Example annotations of schema 1 136
8.2 Example annotations of schema 2 136
8.3 Generated output for the detection of semantic changes 142

Chapter 1

Introduction

[1] Interoperability between enterprises, in particular document exchange and inter-organizational business processes [33] is a key problem in enterprise integration. XML and Web Service technologies support syntactic interoperability. They allow the manual implementation of interoperating services with comparable low setup-costs. Therefore, these technologies are well suited for interoperability on the syntactic level but they do not address the semantic level which is therefore, realized manually by programmers. Technologies from the Semantic Web address this issue by adding semantics to allow interoperable applications on the syntactic and the semantic layer. It is hoped that this will further reduce the setup costs for interoperating services. However, the implementation of interoperability software is a considerable effort. A limiting factor for the return of this investment is the expected life-time of interoperability layers. A major problem for partners establishing interoperating services is the evolution of the participating information systems. Here maintenance of interoperability software is a major concern. Since partners - respectively their information systems - constantly undergo changes to improve and adapt to new constraints, regulations or requirements, the software layers have to be changed as well. This typically causes maintenance costs which can be higher - over the lifetime - than the initial costs for the setup of the interoperating application. This fact has another consequence: The evolution in heterogeneous systems which should take place to adapt to new requirements is often not realized because of the expected adaption costs. Semantic Web Technologies can solve this evolution problem intrinsically as long as the semantic level stays unchanged and changes appear on the syntactic level. The semantic level does not change as long as the domain model (ontology) does not change. Unfortunately, changes in the domain are typical reasons for the evolution of information systems. Therefore, changes on the semantic level must be addressed.
Another constraint for the successful application is that the solutions have to scale up to high volume interactions as they are frequent in enterprise integration. From the perspective of the Semantic Web instance data are represented on the semantic level such as RDF [66] or

[1] Parts of this chapter have been published in [24]

OWL [19] instances. Unfortunately, computation on the semantic level is usually extremely expensive due to reasoning and therefore, does not scale up for enterprise-scale applications. On the one hand we suppose that XML technologies are and will be the key technologies in enterprise-scale applications and that they will not be substituted by technologies from the Semantic Web in the near future. On the other hand the latter technologies are well-suited to express semantics. In such a scenario semantic annotation can be used to get the best of both worlds: The syntactic format is still XML and semantics are expressed with annotations. Semantic annotations are already used for document reconciliation in mediator architectures [64] [35]. They can perform as a driver for interoperability because they can drastically reduce the mapping afford between interoperating systems. Nevertheless, there are different types of semantic annotations. On the one hand there are purely declarative annotations and on the other hand there are mappings that basically transform XML-instance data to their semantic representation. The latter ones allow a maximum on flexibility but we claim that their application has a negative impact on the scalability as well as on the maintainability, when the ontology evolves.

Therefore, we propose an architecture for interoperability in an evolving environment that intends to overcome the described limitations. It focuses on the problem, that XML-business documents need to be transformed before they can be exchanged between different partners because they use heterogenous XML-Schemas. Those schemas are annotated with a common reference ontology. Since the ontology is not assumed to be static changes in the ontology can have impact on the annotations and the semantics of the instance documents. In such a scenario the ontology gets an additional function: It does not only model a snapshot of the real-world domain at some point in time - it also expresses the changes. This can be valuable information for the adaption of the information systems of the involved parties.

The contributions of this research are the following:

- A declarative semantic annotation method for XML-Schemas.
- Schema mapping methods based on the proposed annotation method.
- Methods for the proper representation of ontology changes.
- Methods for the structural and logical maintenance of annotations when the ontology evolves.
- Detection methods for semantic changes after ontology evolution.

The aim of this thesis is pure research that identifies the required sub-problems and proposes solutions. It should be a basis for the future implementation of a tool-set. We will describe the contributions in more detail in section 1.1 and provide a case study in chapter 8 that aims to demonstrate how the different contributions of this research can be used together in a possible use-case.

Publications: The results of this research were published in numerous publications. We have published the general problem of semantic annotations of XML-Schemas in an evolving environment in [24]. The declarative annotation method for XML-Schemas was published in [57]. The implementation of the annotation based schema mapping approach was published in [96]. In [58], we have published the change-representation method and the detection of semantic changes. The detection of logical invalidations was published in [59].

1.1 Contents of the Thesis

In this section we will briefly describe the chapters of this thesis. This description is aimed for a reader who is well schooled in the areas of interoperability and ontologies. For unexperienced readers in these areas we suggest to read section 1.2 which shortly introduces the terms first.

1.1.1 Semantic Annotation

An XML-Schema does not define the semantics of the schema elements (see section 1.2.2). One way to allow semantic interoperability (see section 1.2.1) is to annotate XML-Schemas with a reference ontology. A W3C recommendation for semantic annotations is SAWSDL [56]. It focuses on the semantic annotation of web service descriptions WSDL [5] but also allows the annotation of arbitrary XML-Schemas [100]. The standard defines two different kinds of annotations. On the one hand declarative annotations that link schema elements to concepts of the reference ontology and on the other hand transformation scripts that transform instance data to ontology instances and vice-versa. Such a transformation script can for example technically be realized by XSLT-Scripts [100] that transform the XML-data to RDF-Data [66] and vice-versa. In this research we focus on purely declarative annotations. We have discovered that the declarative annotations that are proposed by SAWSDL do not provide the proper expressivity and have therefore, proposed an enhanced method that is still purely declarative but allows the description of the schema elements in more detail. The complete annotation method can be found in chapter 2.

1.1.2 Schema Mapping

A schema that is annotated with a reference ontology can be mapped to another schema that is annotated with the same ontology by exploiting the explicitly defined semantics of the schema elements. The main purpose of such a mapping in this research is the generation of data transformations between the different schemas. There are two main approaches on how data-transformations that are based on semantic annotations can be realized. First of all, there is the lifting/lowering approach. In this case the annotations are scripts that transform

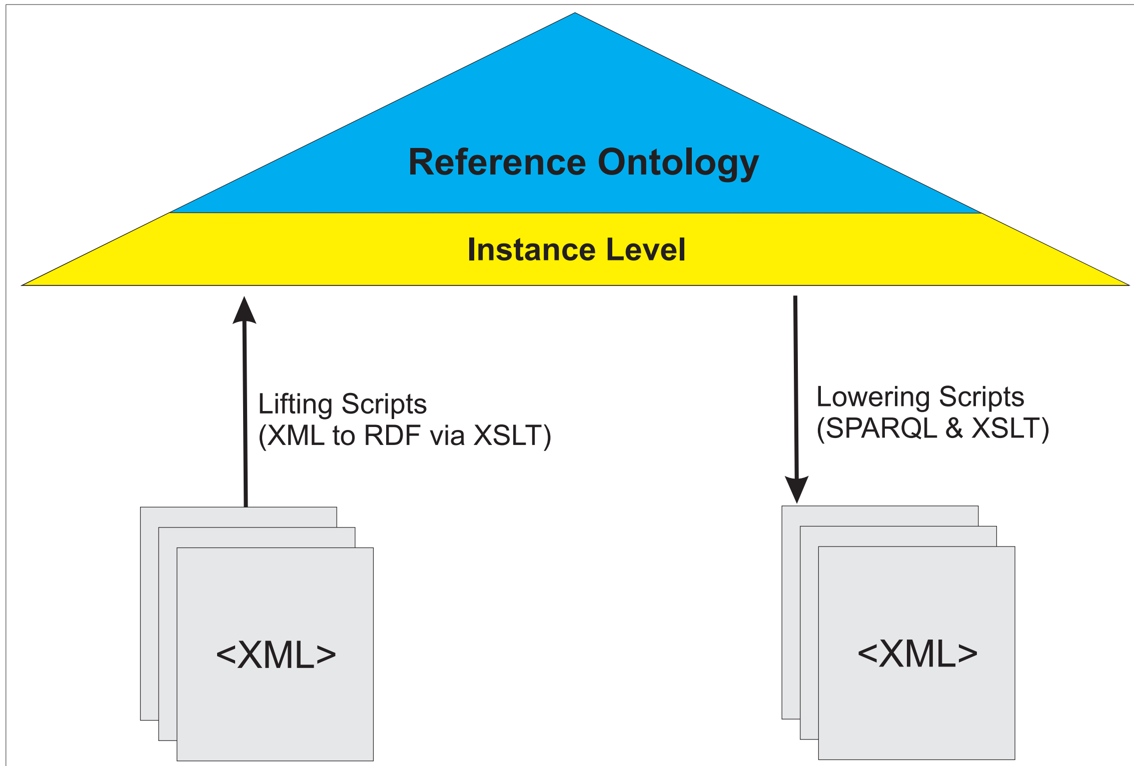

Figure 1.1: XML document transformations using the lifting/lowering approach

the XML-instances to instances of the reference ontology and vice-versa. When documents are transformed between two XML-Schemas they are first lifted to ontology instances, then processed on the ontology level using standard reasoning methods and rule languages and are finally lowered to documents of the target XML-Schema. The overall scenario of such an approach is depicted in figure 1.1.

We propose another approach based on declarative annotations that allows the generation of schema mappings. Those mappings can be used to generate scripts e.g. XSLT [15] that directly transform instance documents from the source schema to instances of the target schema. The overall approach is depicted in figure 1.2. We propose that this approach has several advantages over the lifting/lowering approach. First of all, the schema matching and transformation script generation is only required once at build-time. After that the XML documents can be processed on the XML-level without using the reference ontology at all. This should have an enormous effect on the scalability of the approach. In addition we suppose that declarative annotations that are based on a well-defined annotation method have the advantage that when the ontology evolves possible repair actions to maintain the annotations can be generated. We will discuss mapping methods and a prototype implementation for XML-Schemas that are annotated with our proposed annotation method in chapter 3. This

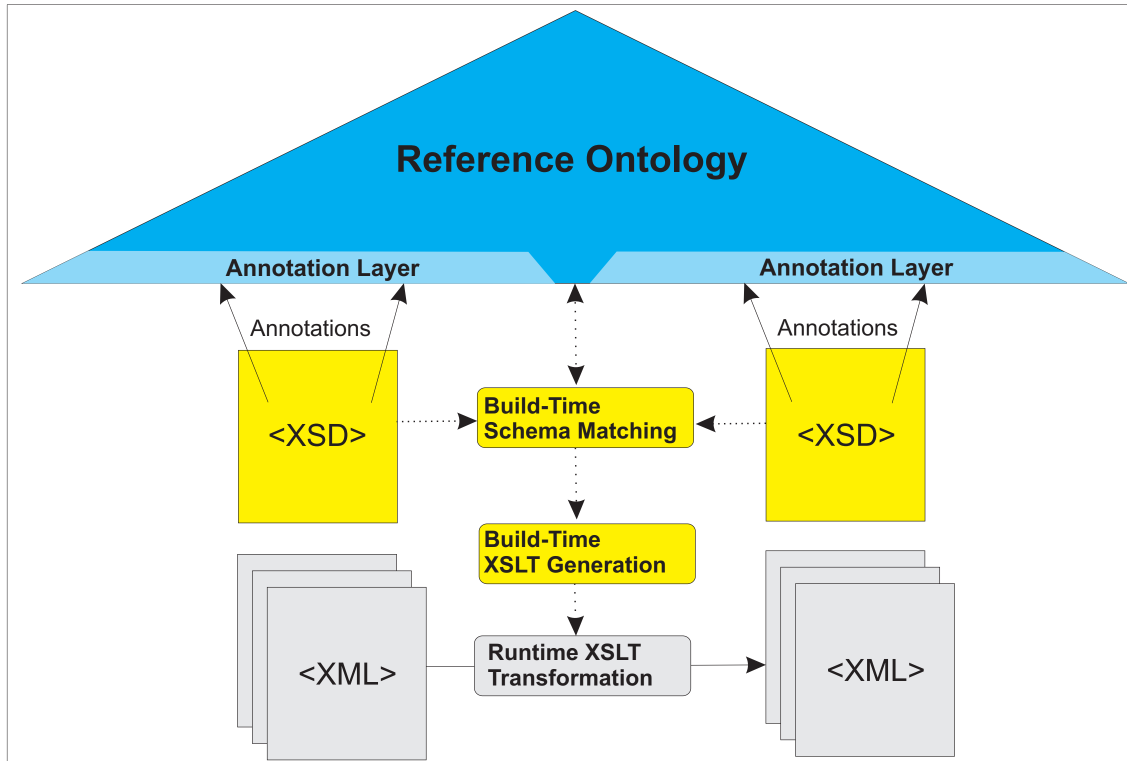

Figure 1.2: XML document transformations using annotation based schema-mapping

chapter also presents a performance evaluation of our prototype implementation against an implementation of the lifting/lowering approach. In addition to a pure lifting and lowering approach there also exist hybrid approaches, where lifting/lowering scripts are generated with the help of declarative annotations [98]. This is not in the scope of this research.

1.1.3 Change Representation

Our aim is to maintain the annotations and to detect semantic changes, when the ontology evolves. This requires a declarative representation of ontology changes. The representation of changes depends on the used ontology formalism. In addition to a representation of the changes that occurred, it is also necessary to allow reasoning support over the consequences of the changes. This is especially necessary for the detection of semantic changes.
In chapter 4 we will first of all, discuss the requirements for change-representation and provide a survey on related work in the field of ontology evolution. Finally, we present our change-representation approach that can be used for the structural and logical annotation maintenance and for the detection of semantic changes using standard Semantic Web technologies.

1.1.4 Annotation Maintenance

When the ontology evolves the annotations of the schemas need to be maintained. This maintenance requires change operations over annotations. There are multiple levels on which an annotation can get invalid with regard to a new ontology version. First of all, it can get structurally invalid. An annotation is structurally invalid, when the annotations does not still comply with the structural requirements of the annotation method. An example for a structurally invalid annotation is an annotation that references a non existing ontology element. Such an error can automatically be repaired, when the change-representation contains information about what has happened to the missing ontology element. We will present change operations and methods for the structural repair of annotation in chapter 5.
Another type of invalidations are logical invalidations. An annotation is logically invalid, when it contradicts with the reference ontology. In this case it is important to detect what element(s) of the annotations are responsible for the logical clash. We will provide an in depth analysis of logical invalidations and algorithms and methods to detect the causes for the invalidations in chapter 6.

1.1.5 Detection of Semantic Changes

In case of structural and logical invalid annotations the elements that needs to be maintained are the annotations. In contrast in case of semantic changes the semantics of the real-world has changed in a way that it has consequences for the instance data. This may require to transform the instance data in order to comply with the new ontology version. We will illustrate the problem with a small example: We suppose there are XML documents of different years that contain data about the population of the European Union. The schema and the annotations were never changed and thus, no annotation maintenance is required. This does not mean that the population data of the different documents can be compared. In fact they cannot be compared because additional countries joined the European Union. We need methods to detect such changes in order to warn the user about changes that have consequences for the interpretation of instance data. An approach that is based on explicit dependency definitions over the ontology elements is proposed in chapter 7.

1.2 Preliminaries

In this section we will briefly introduce interoperability, XML-Schema and ontologies because the terms are extensively used during the rest of the thesis.

1.2.1 Interoperability

The IEEE defines interoperability as *the ability of two or more systems or components to exchange information and to use the information that has been exchanged [74].* This definition contains two requirements: First the information needs to be exchanged in a format that allows the exchange of information between different systems. This is typically called syntactic interoperability and can for example be realized in form of messages that are represented in form of XML [6] documents. The second part of the definition states that the systems must be able to use the exchanged data. This requires that the meaning of the data is known to the interoperating systems. It is typically considered as semantic interoperability. This can for example be achieved by the usage of a common data-model. This can be a standard data-model or an ontology that defines the semantics of the application domain.
When we assume that the world is non static interoperability is not only required between different information systems but also between different versions of one system. Thus, interoperability with regard to time allows to correctly process data that was created with a different version of the same information system.

1.2.2 XML-Schema

As described in the last subsection XML can be used as a basis for syntactic interoperability. XML alone only defines the markup language that is used to exchange documents. In order to process the exchanged messages the structure of the messages needs to be defined. The most important standard to define the structure/format of XML documents is XML-Schema [100]. An XML-Schema is itself an XML document. It consists of definitions for elements, attributes, simple types and complex types. There exists a predefines set of simples types (such as string, integer, ...) which can be extended by the user. In contrast to simple types, complex types are used to define structures that consists of sets of elements. XML allows to reference elements or types from other elements which can be used to reuse definitions. When multiple partners agree on a common XML-Schema for the exchange of messages their software systems can rely on the format of the exchanged XML-documents. This allows the systems to correctly parse the data and enables syntactic interoperability. In contrast XML-Schema does not describe the semantics of the schema elements in a machine interpretable form. Thus, is does not allow semantic interoperability. In this research we use semantic annotations with a reference ontology to define the semantics of the schema elements.

1.2.3 Ontologies

As described in subsection 1.2.1 ontologies can act as a driver for semantic interoperability. The simplest form of an ontology is a taxonomy. It is organized in form of a simple tree structure that defines terms of some real-world domain. Examples for taxonomies are biological

classifications. In a taxonomy we could for example state that tigers and leopards are carnivore by representing tiger and leopards as child-nodes of carnivore. In addition to such a tree representation there is no additional description of the nodes and there is typically only one relation-type allowed. In our example it is a *isA* relation that states that all tigers and leopards are carnivores. A taxonomy has very limited expressiveness. However, in many application domains taxonomies can be used as a powerful source of knowledge.
In contrast to a taxonomy a general ontology allows the description of the domain in much more detail. According to [36], *An ontology is a formal explicit specification of a shared conceptualization*. An ontology typically defines concepts, properties and individuals of some domain. Concepts are also often referred as classes, properties are often referred as roles and both are structured in subsumption hierarchies. The hierarchies are not limited to tree structures. A concept is typically described by defining its extend. The extent is the set of individuals that are members of that class. This definition can be realized by defining the properties that individuals of a specific class have. For example we can state a person is a human being and it has a birthdate, where the property birthdate connects an individual of the class person to some date. In addition a person has the (object) property *hasMother* that links an individual of a person to another individual of a person. According to the used modeling formalism additional semantical constructs are allowed to define concepts and properties more precisely. Nevertheless, hierarchies of classes and properties build the explicit backbone of an ontology, but the explicit class or property hierarchy is not necessarily equal to the inferred hierarchy. Ontologies can be based on very expressive and undecidable modeling formalism which limits the practical usefulness. Therefore, for computational purposes typically restricted logical foundations are used for ontologies.

1.2.4 Ontology Formalisms and Languages

In addition to the general types of ontologies there are different modeling formalisms and languages for ontologies which have influence on their expressive power. We will discuss the most commonly used ones Frames, RDF-Schema and OWL description logics in the next subsections.

Frame-Systems

One of the first methods for the representation of knowledge bases and also for ontologies are frame-systems. They were first introduced by Marvin Minsky [67] and are used in numerous knowledge-based systems [30]. The popular ontology editor protege[2] used a frame-based ontology representation until version 3. The basic principle of frame-based systems are frames. A frame can represent objects as well as classes of objects. A frame that represents a class can have multiple superclasses, while a frame that represents an object can be a member of

[2] http://protege.stanford.edu/

multiple classes. A frame that describes a class is just a member of the class *classes*. Therefore, this approach is suitable for meta-modeling. The superclass relation allows the definition of subsumption hierarchies including multiple inheritance. In order to describe the properties of a specific class or object slots are used. Typically classes use *MemberSlots* that define that all objects of this specific class have this slot (the class is a prototype for the objects). In case of frames for objects *ownSlots* are used to express that the specific object has an assertion for a specific property. In addition *ownSlots* can also be used to describe properties that apply to a class itself and not to its objects. A slot can refer to a slot from another frame. For example a *height* slot of a *car* can refer to the *height* slot of *physical objects*. Facets are used to restrict the values of the slots. Typical predefined facets are cardinality (min and max), classes of values and lists of possible values.
Frame-Systems typically allow taxonomic reasoning and the definition of custom production rules in order to build expert-systems. Reasoning typically relies on the unique name assumption (can be an internal unique name with non-unique labels for users) and the closed world assumption. Thus, things that have different identifiers are considered to be different as long as they are not stated to be equal. The closed world assumption has the consequence that everything that cannot explicitly be proofed to be true is considered to be false and only things that are stated to be allowed are allowed.

RDF-Schema

RDF-Schema/RDFS [7] is a very limited ontology language that basically allows to define a class and a property hierarchy. Properties have a domain and a range that defines the classes that are subjects and objects of that property. RDF resources can be defined to be instances of some class. This already allows simple taxonomic reasoning and the addition of inferred class or property assertions to an RDF-graph.

Description Logics and OWL

Another widely used formalism for ontologies are description logics (DL) which are formal decidable fragments of first order logics. They have been used for ontology modeling for a long time and are also the foundation of the ontology standard of the W3C OWL(2) DL [19, 75]. By applying formal semantics they allow to infer additional knowledge. This reasoning support is an important feature for an ontology language. There are different kinds of description logics that have differences in their expressivity. This has consequences on the computational complexity and decidability of the specific description logics. The basic building-blocks of description logics ontologies are: Abox Axioms, Tbox Axioms and Rbox axioms.
Abox axioms assert knowledge to individuals. This can either be concept assertions that define that some individual belongs to some class such as *person(john)* or role assertions such as *friend(john,marc)*. In addition individuals can be stated to be same as other individuals or to

be different from other individuals. **Tbox axioms** relate concepts with other concepts. Typical relations are subclass, equivalent-class and class disjointness.
Rbox axioms define the relations between roles/properties. This can be axioms over the property hierarchy, disjointness axioms or role inclusion axioms (see property chains in OWL in section 6.3.3).
In addition DL-based ontology languages typically support different kinds of constructors for concepts and roles. For concepts this can be union, intersection or negation. In addition concepts can be constructed by role restrictions. Such restrictions restrict roles on concepts. They can be expressed with existential or universal quantification or cardinality restrictions. A (qualified) role restriction on a concept can for example express that each person has exactly one *hasFather* relation to some male person. Finally, classes can be constructed by enumerating their individuals.
There is typically only a limited number of role constructors in ontology languages. An example is the definition of inverse roles. This allows for example to specify that $childOf$ is the inverse role of $parentOf$. In addition property characteristics can be used to characterize roles in form of the typical properties of relations: transitivity, symmetry, asymmetry, reflexivity, and irreflexivity. The definition of the domain and range of a property can logically be expressed by using the top-concept (in OWL *Thing*). For example the domain of $writesExam$ is a $Student$ can be defined as: $\exists\ writesExam.Thing \sqsubseteq Person$, the range $Exam$ can be defined as $Thing \sqsubseteq \forall\ writesExam.Exam$.
OWL DL is a sub-language of OWL that is based on description logics. The described properties of DL-languages for ontologies directly comply with the formal grounding *SROIQ* of OWL2 DL. We will not go into detail for this work and refer the interested reader to [60]. Most additional constructs of OWL-DL are just syntactic sugaring and can be mapped to DL. OWL reasoning is based on the absence of the unique name assumption and uses the open world assumption. [60]

Chapter 2

Semantic Annotation

[1] Semantic annotation is proposed to be a good solution to enable interoperable applications [97]. Semantic annotations represent the relationships between an annotated artifact (web page, XML document, schema, web service, etc.) and a reference ontology. Semantic annotation at the instance level received a lot of attention [97], however annotation at the XML-Schema level [56] is used in a much lesser degree. Semantic annotations of XML-Schema can be used to lift data from XML documents to some semantic representation such as RDF [66] or OWL [19] instances. Therefore, a transformation of a document from a source XML-Schema to a document that complies with the target XML-Schema can be created by lifting the data from the source document to its semantic representation, (e.g. ontology instances) make some computations on the ontology-level and lower it back to the XML representation of the target schema. The scenario is shown graphically in figure 1.1. Such an approach is very flexible and powerful on the one hand but requires expensive semantic processing of every instance document on the other hand. Therefore, we propose to generate transformation scripts with knowledge from the ontology to achieve industry-scale performance. This requires the annotation of the source and the target schema in a declarative way. Such annotations allow the matching of the source and the target schema at build time (see figure 1.2). The resulting mapping can then be used to create transformation scripts [42] (e.g. XSLT) that directly operate on the XML documents without the need to lift instance data to the ontology.

Both approaches are addressed in the W3C recommendation SAWSDL [56]. The lifting and lowering approach is realized by the specification of references to arbitrary scripts that perform the lifting or lowering of instance data. The declarative annotations can be realized by so called *model-references*. A *model-reference* forms a relation between a schema element (XML-element, XML-type or XML-property declaration) and a concept of some semantic model. In this chapter we will investigate the applicability of SAWSDL model-references for the declarative annotation of XML-Schemas with a reference ontology. We will discuss shortcomings

[1] Parts of this chapter have been published in [57]

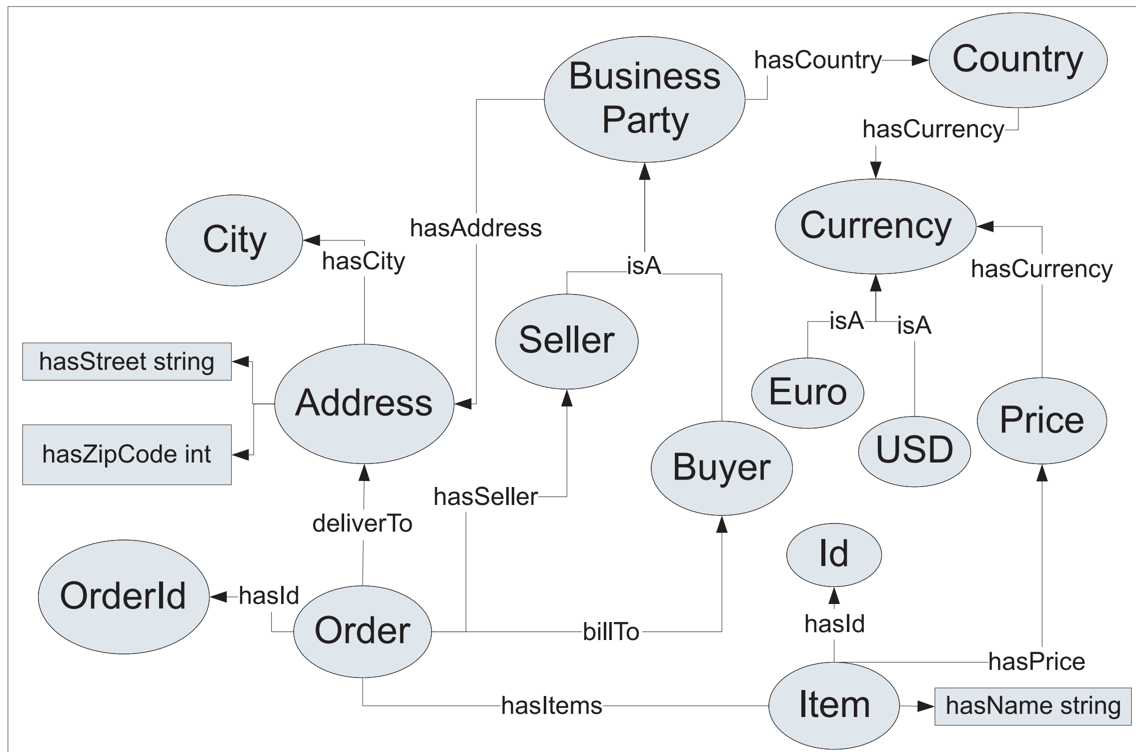

Figure 2.1: Example reference ontology

and present an annotation method that solves these problems while being compatible with SAWSDL. These annotations are the basis for the schema mapping and transformation generation approach in chapter 3.

2.1 Motivating Example

In order to show shortcomings of plain model-references we will first introduce an example. We will try to use model-references for the direct annotation of a simple XML-Schema of a business document shown in figure 2.2 with a small reference ontology that is shown in figure 2.1. The domain of a SAWSDL model-reference is an XML-element, XML-type or XML-attribute declaration of an XML-Schema. The range is a list of URIs that point to concepts of a semantic model. If multiple URIs are specified, every URI applies to the annotated element. No further relationships between the different URIs can be specified.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:sawsdl="http://www.w3.org/
  <xs:element name="order" sawsdl:modelReference="/order">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="BuyerZipcode"/>
        <xs:element name="BuyerStreet"/>
        <xs:element name="BuyerCity" sawsdl:modelReference="City"/>
        <xs:element name="BuyerCountry" sawsdl:modelReference="Country"/>
        <xs:element name="SellerCountry" sawsdl:modelReference="Country"/>
        <xs:element name="Item" maxOccurs="unbounded" sawsdl:modelReference="Item">
          <xs:complexType>
            <xs:attribute name="ID" use="required" sawsdl:modelReference="Id"/>
            <xs:attribute name="Name" use="required"/>
            <xs:attribute name="Price" use="required" sawsdl:modelReference="Price"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Figure 2.2: Example XML-Schema with standard model-references

In figure 2.2 an example *order* document is shown. It is directly annotated with the reference ontology (see figure 2.1). We will now investigate whether the correct semantics of each element can be defined.

- The element *BuyerZipcode* could not be annotated at all because the *zip-code* is modeled in form of a datatype-property and not by a concept in the ontology. The same problem exists for the *BuyerStreet* element and for the name of an *item*.
- The *BuyerCountry* element is annotated with the concept *country*. This does not fully express the semantics because we do not know that the element should contain the country of the buying-party. In addition the *SellerCountry* element has exactly the same annotation and can therefore, not be distinguished.
- The attribute *Price* is annotated with the concept *Price*. Unfortunately this does not capture the semantics. We do not know the subject of the price (an item) and we do not know the currency.

In the examples above we assumed that we have only annotated data-carrying elements. If we would in addition annotate the parent elements in this case the *order* element we could add a bit more semantic information. It would be clear that the annotations of the child-elements of the order-element can be seen in the context of an *order*. Unfortunately this would not help for

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:sawsdl="http://www.w3.org/ns/sawsdl" elementFormDefault="qualified" attril
  <xs:element name="order" sawsdl:modelReference="/Order">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="BuyerZipcode" sawsdl:modelReference="/Order/deliverTo/Address/hasZipCode"/>
        <xs:element name="BuyerStreet" sawsdl:modelReference="/Order/deliverTo/Address/hasStreet"/>
        <xs:element name="BuyerCity" sawsdl:modelReference="/Order/deliverTo/Address/hasCity/City"/>
        <xs:element name="BuyerCountry" sawsdl:modelReference="/Order/billTo/Buyer/hasCountry/Country"/>
        <xs:element name="SellerCountry" sawsdl:modelReference="/Order/hasSeller/Seller/hasCountry/Country"/>
        <xs:element name="Item" maxOccurs="unbounded" sawsdl:modelReference="/order/hasItems/Item">
          <xs:complexType>
            <xs:attribute name="ID" use="required" sawsdl:modelReference="/Order/hasItems/Item/hasId/Id"/>
            <xs:attribute name="Name" use="required" sawsdl:modelReference="/Order/hasItems/Item/hasName"/>
            <xs:attribute name="Price" use="required" sawsdl:modelReference="/Order/hasItems/Item/hasPrice/Price[hasCurrency/Euro]"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Figure 2.3: Example XML-Schema with the proposed annotation method

the ambiguities between the *BuyerCountry*- and the *SellerCountry* element. In general, it would require a very strong structural relatedness between the ontology and the annotated XML-Schema which we cannot guarantee when many different schemas are annotated with a single reference ontology. In addition SAWSDL does not define that there are any relations between the annotations of parent and child elements. Nevertheless, such annotations could help to provide additional knowledge to structural XML-matching methods such as [85]. Another solution is the usage of a more specific reference ontology which contains concepts that fully match the semantics of each annotated element. For example it would need to contain the concept *InvoiceBuyerCountry* and *InvoiceBuyerZipCode*. Enhancing a general reference ontology with all possible combinations of concepts leads to a combinatorial explosion. This is definitely not suitable for the annotation with a general reference ontology but can nevertheless, be used if very specific ontologies are used for the annotation.

2.2 Annotation Method

As shown in section 2.1 the direct usage of model-references is not suitable for the annotation of XML-Schemas with a reference ontology. To overcome this shortcoming we propose to create the required more specific ontology concepts out of well defined path expressions at runtime of the schema-matching engine. We will first introduce the path expressions with some examples and provide the formal definitions in section 2.2.1.

The example schema document in figure 2.3 is annotated with the proposed path expressions. We will discuss some examples: The element *BuyerZipcode* is annotated with ***/Order****/deliverTo/****Address****/hasZipCode*. The annotation of the *BuyerCountry* element is

/Order*/billTo/****Buyer****/hasCountry/****Country***. The steps that are marked bold refer to concepts. The other steps refer to object-properties or datatype-properties of the reference ontology. Now the *BuyerCountry* element can clearly be distinguished from the *SellerCountry* element and the elements *BuyerZipcode* and *BuyerStreet* can be annotated. The shown paths refer to concepts, object properties and datatype properties. Another requirement could be to address instances of the ontology. For example the path ***/Order****/billTo/****Buyer****[Mr_Smith]/hasCountry/****Country*** defines that the Buyer is restricted to one specific buyer with the URI *Mr_Simth*.
In most cases we assume that a simple annotation path as shown in the examples above is sufficient for an annotation. Nevertheless, there can be cases where additional restrictions are required: When using a simple path expressions as shown above the *Price* attribute of the example schema can be annotated with ***/Order****/hasitems/****Item****/hasPrice/****Price***. Unfortunately, this does not express the currency of the price. Since the example ontology has no specialized price-concept for each currency we need to define the price within the annotation. The correct currency of a price can be defined by a restriction on the price concept. This restriction is denoted in square brackets and expresses that the price must have a *hasCurrency* property that points to the concept *Euro*. This leads to the full annotation of the *Price* attribute: ***/Order****/hasitems/****Item****/hasPrice/****Price****[hasCurrency/****Euro****]*.

2.2.1 Formal Definition of the Annotation Method

In order to define the annotation method we will first introduce definitions for the reference ontology and an annotated schema.

Definition 1. ***Ontology:***

An ontology O is a tuple $O = (C, DP, OP, I, A)$, where C is a set of concepts (also often referred as classes), DP as set of datatype-properties, OP a set of object-properties, I a set of individuals and A a set of axioms over C, DP, OP, and I. Each element in C, DP and OP is a tuple $(uri, definition)$. All URIs of concepts can be obtained by $C.uri$, $URIs$ of properties by $DP.uri$ and $OP.uri$ and URIs of instances by $I.uri$ respectively.

Definition 2. ***Annotated XML-Schema:***

An annotated XML-Schema S is a tuple $S = (T, E, A)$, where T is a set of types, E a set of elements, and A is a set of semantic annotations. An XML-Schema forms a tree structure. Each $t \in T$ has a type $e.type = \{simple \mid complex\}$, an optional name $t.name$ and an optional SAWSDL mode reference $t.annotation \in A$. An element $e \in E$ can have an optional type $e.type \in T$, an optional SAWSDL model-reference $e.annotation \in A$ and a set of attributes $e.attribute$. Each attribute $a \in e.attribute$ can have a simple type $a.type \in T$ and an optional model-reference $a.annotation \in A$. Each annotation must be a valid annotation path according to definition 3 and 4.

Definition 3. ***Annotation Path:***

The set of all annotation path expressions is P. An annotation path $p \in P$ is a sequence of steps. Each step is a tuple s=(*uri*, *type*, *res*). The value $s.uri$ of a step is some URI of an element of the reference ontology O. The type $s.type$ can be cs for a concept-step, op for an object-property step or dp for a datatype-property step. The URI $s.uri$ determines the type of the step: $s.uri \in C.uri \Rightarrow s.type = cs$; $s.uri \in OP.uri \Rightarrow s.type = op$; $s.uri \in DP.uri \Rightarrow s.type = dp$. Only concept-steps may have a set of restrictions $s.res$. Each restriction $\in s.res$ can either be an individual $\in I.uri$ or a restricting path expression. Such a path expression adds a restriction to the corresponding step s. If $s.res$ contains multiple restrictions they all apply to the corresponding step s (logical and). The succeeding step of s in p can be obtained by $s.succ$, the previous step by $s.prev$. The first step of p is denoted f_s and the last step l_s. Each annotation path $p \in P$ has a type $\in \{ConceptAnnotation, DataTypeProperyAnnotation\}$. The type of the path is defined by the type of the last step.

Definition 4. ***An annotation path is structurally valid iff:***

- $f_s.type = cs$ - The first step must refer to a concept.
- $l_s.type = \{dp|cs\}$ - The last step must refer to a concept or a datatype-property.
- $\forall s \in p|s.type = cs \wedge s \neq l_s \Rightarrow s.succ.type = \{dp|op\}$ - The successor of a concept-step must be an object-property or datatype-property step.
- $\forall s \in p|s.type = op \Rightarrow s.succ.type = \{cs\}$ - An object-property step must be followed by a concept-step.
- $\forall s \in p|s.type = cs \wedge s \neq f_s \Rightarrow s.prev.type = op$ - The previous step of a concept-step must be an object-property step (except the first step).
- $\forall s \in p|s.type = op \Rightarrow s.prev.type = cs$ - The previous step of an object-property step must be a concept-step.
- $\forall s \in p|s.type = dp \Rightarrow s = l_s$ - Only the last step can refer to a datatype-property.

2.2.2 Reuse of Global Types or Elements

If the annotated XML-Schema reuses types or elements (via type or ref properties) and both the element and the referenced element or type are annotated then the semantics need to be constructed based on the annotation of the element and the annotation of the referenced element. Due to the hierarchical structure of XML this needs to be applied recursively.
Let e be an element with the annotation $e.annotation$ and the XML-Type $e.type$. Let s be an annotated sub-element of the XML-Type $e.type$, then the complete path of s needs to be

constructed by combining the annotation *e.annotation* and *s.annotation*. In particular the combination is achieved by removing the last step of *e.annotation* and concatenating it with *s.annotation*.
As an example we may have an XML-element called *DeliveryAddress*. It is itself annotated with the annotation path ***/Order**/deliverTo/**Address***. It has a type definition *address*. The address type itself contains various elements. One of them is *street* which is annotated with ***/Address**/hasStreet*. In order to construct the complete semantics of the *street* element that has the parent element *DeliveryAddress* we need to build the path ***/Order**/deliverTo/**Address**/hasStreet*. This path combination needs to be performed by the schema matching engine. It must be noted that this path combination adds structural dependencies between the schema and the reference ontology. Therefore, one XML-Type should only be reused for semantically related entities.

2.3 Transformation of Annotation Paths to Ontology Concepts

In the last section we have defined an annotation path expression as a sequence of steps. In order to specify the semantics of such a path expression it is represented in form of an ontology concept. This allows concept-level reasoning over the annotated elements in order to assist the matching of schema elements. The name/URI of such a concept is the corresponding path expression and can therefore, directly be used as a SAWSDL model-reference. OWL allows to define concepts with logical expressions in form of restrictions over its individuals. We will illustrate the generation of concepts for annotation paths with examples for the generation of concept annotation and datatype annotations.

```
1 Class: Order/billTo/Buyer[Mr_Smith]/hasCountry/Country
2 EquivalentClasses(
3       ConceptAnnotation and Country and inv
4         (hasCountry) some
5            (Buyer and {Mr_Smith} and inv (billTo) some (Order)
6         )
7 )
```

Listing 2.1: Representation of a concept annotation path in OWL

In listing 2.1 the OWL representation of the path ***/Order**/billTo/**Buyer**[Mr_Smith]/hasCountry/* ***Country*** is depicted. It creates a specialization of a *country* concept. In particular a *country* that has an inverse *hasCountry* object-property to a *Buyer*. This buyer must be an individual of the enumerated class {*Mr_Smith*} and must have an inverse *billTo* relation to an *Order*.
Obviously such a translation can be achieved fully automatically by iterating over the steps of the path. The example in listing 2.1 shows the transformation of a concept annotation to an ontology concept. It is required to semantically separate concept annotations and datatype-property annotations. Therefore, concept annotations are subconcepts of the special

concept *ConceptAnnotation*. Datatype annotations are subconcepts of the special concept *DataTypePropertyAnnotation*. An example for such an annotation is shown in listing 2.2.

```
1 Class: Order/billTo/Buyer/hasAddress/Address/hasZipCode
2 EquivalentClasses(
3     DataTypePropertyAnnotation and Address and hasZipCode some Literal
4     and inv
5        (hasAddress) some
6           (Buyer and inv (billTo) some (Order)
7        )
8 )
```

Listing 2.2: Representation of a datatype annotation path in OWL

The generation of the concepts can be realized by the schema-matching engine. The generated concepts are only required during the generation of the matching. In general a standard annotation path as shown in the examples always creates a subconcept of the last concept-step. Therefore, the inverse of the object properties is used. Annotation paths p that are used in restrictions $p \in s.res$ of some concept-step s always create subconcepts of the corresponding concept with the URI $s.uri$. Thus, the object-properties are directly used.

2.4 Validation of Annotations

In the last sections we have defined the structure of an annotation path and have shown how an annotation path can be transformed to an OWL concept. This does not guarantee that the generated concepts do not introduce contradictions to the ontology. As an example we may have a path: ***/Order**/deliverTo/**PoBox*** and the ontology defines that the *deliverTo* property may never point to a post office box. When this path is represented as an OWL concept it can never contain individuals and thus, introduces contradictions to the ontology.
In addition the ontology may contain datatype restrictions that restrict values of datatypes to specific types such as string or integer. If such restrictions exist in the ontology they must also exist in the schema. The constraints in the schema must be at least as restrictive as in the ontology. The considerations above lead to the definition of the consistency of an annotated schema:

Definition 5. *A schema S and a set of annotation paths P are consistent with an ontology O iff:*

1. Every annotation path $p \in$ P is structurally valid (see definition 4).

2. Every annotation path $p \in P$ is logically valid (can be expressed as a satisfiable OWL-concept in O).

3. All annotated elements in S are more or equally constraining the values as the corresponding datatype properties in O.

Obviously these requirements can be checked fully automatically: The first check can be realized on the structural level. Each referenced concept and property must be a concept or property of the reference ontology and the restrictions from definition 4 must not be violated. The second check is a typical reasoning task that can be done by any OWL reasoner. Check 3 can be realized by traversing the schema and querying the restrictions from the ontology.

2.5 Related Work

In contrast to the annotation of web resources there is only a small number of related work in the field of the annotation of XML-Schemas. To the best of our knowledge there is no comparable approach for a formal definition of declarative semantic annotations for XML-Schemas that allows class-level reasoning over the annotated elements. We see the application of the annotations in the possibility to create more precise schema matchings than traditional approaches [85]. When a schema matching/mapping can be found transformations can be created that transform the instance documents on the XML-level [42]. In [10] an annotation method and tool for RDF-Schemas [7] is presented. It is based on a general work on annotations and mismatches in [68]. A case-study, where the tool is used is presented in [98]. The idea is to annotate an RDF-Schema with a reference ontology in order to (semi-automatically) generate reconciliation rules that transform the instances of the source schema to instance of the reference ontology and vice-versa. The approach does not directly operate on XML-Schemas, thus the XML-Schemas and documents need to be transformed to RDF-Schema/RDF first. The annotation expressions are also represented in form of OWL concepts and are based on path expressions that can be compared to our annotation path expressions. However, the method also focuses on the runtime perspective allowing to specify abstract operators that are used to define data-transformation templates. This allows to solve mismatches directly on the annotation layer that we need to treat on another layer. In contrast our approach can directly be used for XML-Schemas and focuses on the mapping generation between XML-schemas and not the generation of lifting/lowering mappings that map instance data to/from ontology instances.
In [4] another approach for the annotation of models is proposed. At first a meta-model that is expressed in OWL is created for every type of model (Relational database, XML-Schemas, ...) which should be annotated. Afterwards individuals for a specific schema are created. This means there is a representation of the concrete schema as an instance in the ontology. Annotations are just mappings between the reference ontology and the individuals of the schema. Therefore, this solution is not based on direct XML-level annotations as proposed by SAWSDL.
In [102] an approach is presented that automatically discovers mappings between XML-

Schemas and ontologies with the help of a given set of simple correspondences between the schema and the ontology. It assumes a structural relatedness and the discovered mappings are expressed in form of rules. Since first order logic rules can only modify instances this approach is well suited for a lifting approach that transforms XML-data to ontology instances. In contrast, our method creates ontology concepts that form declarative descriptions which are a basis to build XML-level transformations without the need of lifting instance data to the ontology at runtime. In [50] the differences between ontologies and XML-Schemas are discussed. The authors propose to model the domain via an ontology and transform this specification to an XML-Schema or database schema. In [52] a system is proposed that automatically creates annotations for web service descriptions. It can use a reference ontology but does not need one. If no reference ontology is provided the ontology is created during the approach. The provided annotations are basically enhancements of the schema-elements with vocabulary of the ontology. They do not provide a complete declarative description. Nevertheless, approaches that automatically generate annotations like [102] or [52] can possibly be a basis for the semi-automatic creation of annotations for our annotation method.

2.6 Conclusion

In this chapter we have proposed a method for the declarative semantic annotation of XML-Schemas that enhances the semantic expressiveness of SAWSDL-model-references. The annotation method has two representations. On the XML-level there are well-defined annotation paths that can be added to XML-Schemas by schema designers without deep ontology engineering skills. These annotation paths can automatically be transformed to ontology concepts. These concepts provide a declarative description of the annotated elements and can be used for class-level reasoning over schema elements in order to create mappings between XML-schemas. Such mappings can be used to automatically create scripts that transform instance-data from one schema to another. It must be noted that our annotation method does not directly resolve all possible conflicts [69] between the schemas and the reference ontology. For example if the attribute granularity of the ontology and the schema differs no direct annotation is possible. Only existing concepts in the reference ontology can be used in the annotation path expressions. We propose to solve such heterogeneities with additional ontology concepts and explicitly defined transformation templates (see chapter 3). In addition to our annotation method we have provided mechanisms to check whether the annotations are valid with regard to the ontology. The proposed annotation method is the basis for the mapping generation in chapter 3. The chapters 5 and 6 will propose methods for the maintenance of the annotations when the ontology evolves.

Chapter 3

XML-Schema Matching and Mapping

The goal of the declarative semantic annotation method from chapter 2 is to describe the semantics of the schema elements in detail in order to allow the creation of schema to schema mappings based on the semantics of the schema elements. Given an annotated source schema S and an annotated target schema T we want to find a schema mapping $M(S, T)$ that maps the nodes from S to the nodes of T.
We will first introduce schema matching/mapping in general and present related work. As a next step we will present our proposed mapping model and discuss semantic relations between annotations of the source and target schema in order to find simple and complex matches. Typically there are mismatches between the source and target schema that need to be resolved in order to find a mapping. Therefore, we will discuss how different types of mismatches can be resolved by using the annotation method and the semantic relations between the annotations. Finally we will present our prototype for matching and mapping with the proposed annotation method. A complementary approach for the mapping of schemas with the help of ontologies is the transformation of instance documents to ontology instances (lifting) and back to the target schema (lowering) and thus, to realize the transformation on the ontology layer. We will finally provide benchmark results between our proposed XML-level transformation approach and an implementation of the lifting/lowering approach.

3.1 Introduction

A key problem in data-integration is the matching of different schemas. In general schema matching is not restricted to XML-Schemas. It can be used for any kind of schemas such as relational schemas, object-oriented schemas, ontologies and many more. It is a broad field of research with myriads [21, 2, 63, 26, 23, 11, 22, 79] of approaches. Multiple surveys such as [85], [84] or [92] provide a good overview about the problem and the proposed solutions.
The application of schema matching lies in different fields such as catalog integration, p2p

databases, agent communication, web service integration and many more [92]. The result of schema matching is a set of correspondences between schema elements [85].
Such correspondences can have very brought semantics like "corresponds" or they can provide some expression in order to define the relation between the source and target elements in more detail. These matches are the input for the creation of schema mappings. Depending on the quality of the matching expression the match result is a first step for a mapping or it can already be a mapping between the schemas. Such mappings can then be used for various applications such as the transformations of instance data from the representation of one schema to the representation of another one or to rewrite queries in order to transparently query data with regard to one schema that is structured by different schemas [81].

3.1.1 Schema Matching

We will introduce schema matching using a general mapping formalism for schemas as proposed in [85]. Given one source schema S and a target schema T we search for a schema mapping M. This mapping should map the schema elements according to their real-world semantics. M is a set of mapping elements. Each mapping element is a tuple $(S1, T2, expression)$, where $S1 \subseteq S$ and $S2 \subseteq T$. The expression can be an arbitrary expression. When expressions over scalars are used the expression could for example define that two elements are equivalent (and that thus, the values can directly be mapped) or it can be a function such as the concatenation of elements or some mathematical expression. But the expressions are not limited to expressions over scalars. Depending on the types of matched schemas this can also be $part-of$, $subsumes$ or $overlaps$ relations. In this research the overall goal is the generation of mappings to allow documents transformations. Therefore, we are especially interested in the creation of mapping elements with scalar expressions.

Schema Matching Approaches

After we have defined the goal of schema matching we can discuss different approaches that can be used to generate this output. The main problem in standard schema matching is that the real-world semantics of the schema elements are not defined. Therefore, the algorithms used to match the schemas can only try to guess the semantics based on different knowledge sources. Examples are element/attribute names, the structural organization, constraints, instance data and possibly external sources such as lexicons or thesauri [85, 84, 92].

Match Granularity

There are two different approaches to generate matches: Element-level and structural level matchers [85]. An element-level matcher matches each element separately, while a structural

matcher matches structures / substructures. For example an XML-Schema structural matcher matches a complete subtree at once, while an element-level matcher matches each element separately, while ignoring the structure.

Match Cardinality

An important point is the characterization of the mapping tuples that a matching method can provide. The survey in [85] distinguishes between global and local cardinalities. The global cardinality defines how often an element of the source and target schema can be used in different mapping elements, while the local cardinality defines how many elements from the source and target schema can occur in each single mapping element. While in the most general form of a mapping element (S1,T2,expression) $S1$ and $S2$ are sets of nodes and $expression$ can be any expression, most of the matching solutions impose limitations for $S1$, $S2$ and exp.

- 1:1 $S1$ and $S2$ contain one distinct element.
- 1:n $S1$ contains exactly one element, while $S2$ is a set of elements.
- n:1 $S1$ is a set of elements while $S2$ contains exactly one element.
- n:m $S1$ and $S2$ are sets.

Composite and Hybrid Matchers

Practical approaches for schema matching typically exploit more than one dimension of a schema. This can be implemented with hybrid matching algorithms that combine multiple dimensions in one algorithm. An alternative and more flexible approach is a composite matcher [21], [2]. Therefore, multiple matchers are used independently and produce partial results. Finally the results are aggregated to a final result. One problem here lays in the configuration and parametrization of the aggregation function of the composite matcher because this parametrization has a big impact on the match quality. A specific parametrization can fit very well for some schemas and produce unappropriate results for others. This leads to approaches that do not work with user-defined configuration values. Instead, the parametrization can be based on adaptive, machine learning approaches. An example for such an approach can be found in [23].

A general Matching Approach

A flexible matching architecture that uses the idea of a composite matcher is COMA [21]. In a first step the different schemas are loaded and preprocessed in the system. This preprocessing includes the transformation of the input schemas to a generic schema model. After that a set of matchers is used. Each matcher generates a $n \times m$ matrix filled with confidence levels, where n

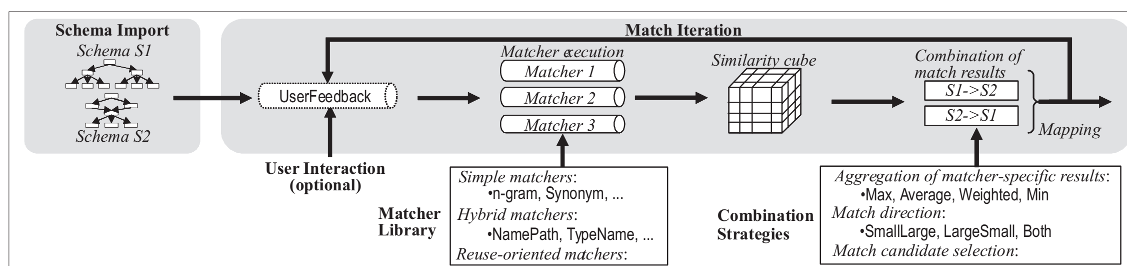

Figure 3.1: Match processing in COMA [21]

and m denote the number of elements in the source and target schema. The confidence levels express the degree of similarity between the matched elements. The result of the matching phase is a $n \times m \times a$ cube, where a is the number of used match algorithms. In a final step the match results are aggregated and combined to a mapping. The overall process is shown graphically in figure 3.1.

3.1.2 Complex Matches

The majority of matching approaches only support the generation of mapping elements with a local cardinality of *1:1* and a *corresponds*-expression [85]. While in many cases this is already useful a system should also support non *1:1* matches and arbitrary functions. Otherwise no complete mapping can be generated. We will now define the matching space for *1:1* and *n:m* matches in order to discuss the different complexity.

Matching Space

Analogues to [21] and its successor [2] we assume that some matching algorithm can produce a $|S| \times |T|$ matrix, where S is the source- and T is the target schema. Each value in the matrix is a confidence value between a schema element of the source and a schema element of the target schema. Obviously this method only allows the detection of a general *corresponds* relation and not a function. Otherwise we would have a cube with the dimensions $|S| \times |T| \times |f|$, where f is a set of functions. So already the addition of more precise matching expressions leads to a much greater matching-space. In addition extra knowledge is required to judge the confidence value for the application of functions. This already big search-space becomes even much bigger, when local non 1:1 cardinalities need to be supported. The search-space for local *n:m* cardinalities is $|\wp(S)| \times |\wp(T)| \times |f|$. Because any subset of elements from the source schema can be matched with any subset of elements of the target schema. Obviously an exhaustive search over this search-space is already unfeasible. In addition it gets even worse, if we do not limit the expressions to exactly one expression, but to a combination of different

expressions. In this case the search-space gets infinite. As a conclusion we can state that even the creation of *1:1* mappings in a standard schema matching case is problematic because the matching of the elements is based on similarities of the structures or the similarity of element names which does not necessarily also mean that they have the same semantics. Composite matchers can provide better matches by exploiting multiple dimensions of the schemas. When it comes to *n:m* matches the potential search space explodes because now powersets need to be matched with powersets. Especially in this case support for matching expressions other than "corresponds to" becomes essential to construct a useful mapping. Unfortunately, the support for arbitrary function results in an even much bigger search-space. Therefore, methods that support *n:m* match cardinalities need additional knowledge and are typically based on heuristics to limit the search-space. Non *1:1* matches with matching expressions other than *corresponds* are also considered as complex matches in this research.

Matching Approaches for Complex Matches

As discussed in the last section the discovery of complex schema matches is much harder than the discovery of standard *1:1* matches. In order to detect such matches additional knowledge is required. As a consequence there are only a few approaches that can deal with complex matches. The DCM framework [39] is not used for schema to schema- but for holistic schema matching. Such an approach, where multiple schemas are matched at once has the advantage that the matching of more schemas also provides more knowledge about the domain. The basic idea is that *n:m* matches can be found by positive and negative correlation mining of the input schemas. For example if it can be discovered that the attribute *name* does never occur together with the two attributes $firstname$ and *lastname* (which both typically occur together) it is likely that $firstname$ and *lastname* can be matched with *name*. The mapping expression is always semantic equivalence. Thus, no lexical functions such as concatenate are supported. At least the result of the matching can be an input for a mapping approach where this information needs to be added manually.
The iMAP System [20] can also discover complex matches. The system is instance based. Thus, the attribute values of instance documents are used for matching. This is realized by multiple searches where each searcher is capable to detect certain kinds of matches. For example the numeric matcher can discover matches that are based on numerical formulas by exploiting the similarity of value distributions. This matcher can for example establish matches such as $price * quantity * (1 + fee - rate)$. Other searchers are schema mismatch searchers, unit conversion searchers and a date-searcher. In contrast to the DCM system iMAP can also return the expression that is needed to map the elements. Nevertheless, it is a semi-automatic task, where the user is assisted by the system to create the final schema mapping. Obviously only matches can be generated that fall into the categories of one of the searchers. In contrast to the previous instance based examples the work in [87] does not use instance-data. Instead it operates on so-called mini taxonomies, which are generated automatically by mining a large set of schemas of the domain. In a first step simple *1:1* matches are generated. Whenever non-leaf

nodes are matched in the *1:1* matching approach, they are candidates for *n:m* matches. The existence of such a *n:m* match is then verified in a next step which uses external knowledge in form of the mini-taxonomies to match the nodes. If a match based on a mini-taxonomy can be established, then the existence of an *n:m* match is considered as validated. This system can propose *n:m* matches but it does not return the specific matching expression.
The work in [28] also uses instance-data to establish a schema mapping. It additionally uses data-frames and domain ontology snippets in order to generate complex schema mappings. A data frame represents the essential properties of data items. In addition to an abstract data-model it also provides information on how to classify instance data. This can for example be realized by regular expressions or by value-lists. Thus, a data-frame can classify instance data to be data of some specific data-frame. Domain ontology snippets describe small aggregates of information. For example an ontology snippet for an address could state that an address consists of the attribute *street*, *city* and *zip*. Finally, data-frames, ontology snippets and the usage of instance data provide additional knowledge to generate complex mappings.

Approach	**Method and Source of Information**	**Mapping Exp.**
DCM Framework [39]	holistic schema matching/correlation mining	no
iMAP System [20]	instance based, specific searchers	yes
Complex schema match discovery and validation through collaboration [87]	mini taxonomies, generated by mining a large set of schemas of the domain	no
Automatic direct and indirect schema mapping [28]	instances, data-frames and domain ontology snippets	yes

Table 3.1: Matching Approaches for Complex Matches

We have summarized the different approaches for the generation of complex schema matches in table 3.1. The table compares the approaches based on the dimensions *method and source of information* and the possibility to automatically generate complex matching expressions. As a conclusion we can state that complex matches require additional knowledge. The proposed systems can generate some kind of mappings but they either rely on instance data or they do not provide a possibility to generate a mapping expression. Even if they generate a mapping expression this is limited to certain types of expressions and it is only a guess, which needs to be validated by a user.

3.1.3 Matching Approaches for XML-Schema

Most of the schema matching methods use a generic internal representation in form of schema graphs or trees. Any type of schema which can be transformed to this representation can be matched with those approaches. In addition to these general approaches also numerous approaches that concentrate on XML-Schema were developed. The PORSCHE [88] system

takes numerous source XML-Schemas as inputs and generates a mediated schema, as well as mappings for each source schema to the mediated schema. ASMADE [91, 90] presents a schema matching and mapping approach that is based on the constraints of the source and target XML-Schemas. Correspondences that are detected by some other matching approach are filtered by using the constraints of the XML-Schemas. In [41] a neural-network based semi-supervised matching approach for XML-Schemas is presented. It learns a synthetic similarity measure that is based on numerous existing similarity measures. The work in [1] classifies, reviews and experimentally compares different methods for the computation of similarity of XML-schemas. Their findings include that it is not sufficient to use only one measurement. Instead, multiple measurements should be combined to provide good matches. In [61] a complete matching system is described that generates *1:1* mappings based on name similarity, node similarity, and structural similarity. In contrast to standard matching systems it is optimized for XML-Schemas.

3.2 Annotation based XML-Schema Matching

We have discussed the general problem of schema matching in the last sections. One main problem of general schema matching is that the semantic of the schema elements is not explicitly defined and thus, needs to be approximated by using various dimensions of a schema. In case of schemas that are semantically annotated with our proposed annotation method the meaning of the elements is explicitly defined. Additional dimensions of the schema only need to be exploited in order to comply with the constraints of the schema. While this makes schema mapping much easier for *1:1* matches, complex matches still need additional knowledge. We will present our proposed mapping model and propose methods for the detection of *1:1* and *n:m* matching elements for annotated XML-Schemas in the next sections.

3.2.1 Schema Mapping Model

The goal of schema matching is to create a schema mapping between a source schema S and a target schema T. Since XML schema allows the reuse of types and elements (see section 2.2.2) the schema mapping should not be defined on the plain input schemas. Instead, we use an unfolded representation for the schemas which we call an expanded schema tree. An expanded schema tree is created by replacing each node that references another node by *type*- or *ref*- references with its definition until no further replacement is possible. The schema mapping $M(S, T)$ consists of mapping elements. Mapping elements can be simple-mapping elements, where the elements directly match (copy semantics) or they can be complex mapping elements that map sets of nodes from the expanded source schema tree S with sets of nodes from the expanded target schema tree T. Those mappings require an expression that defines

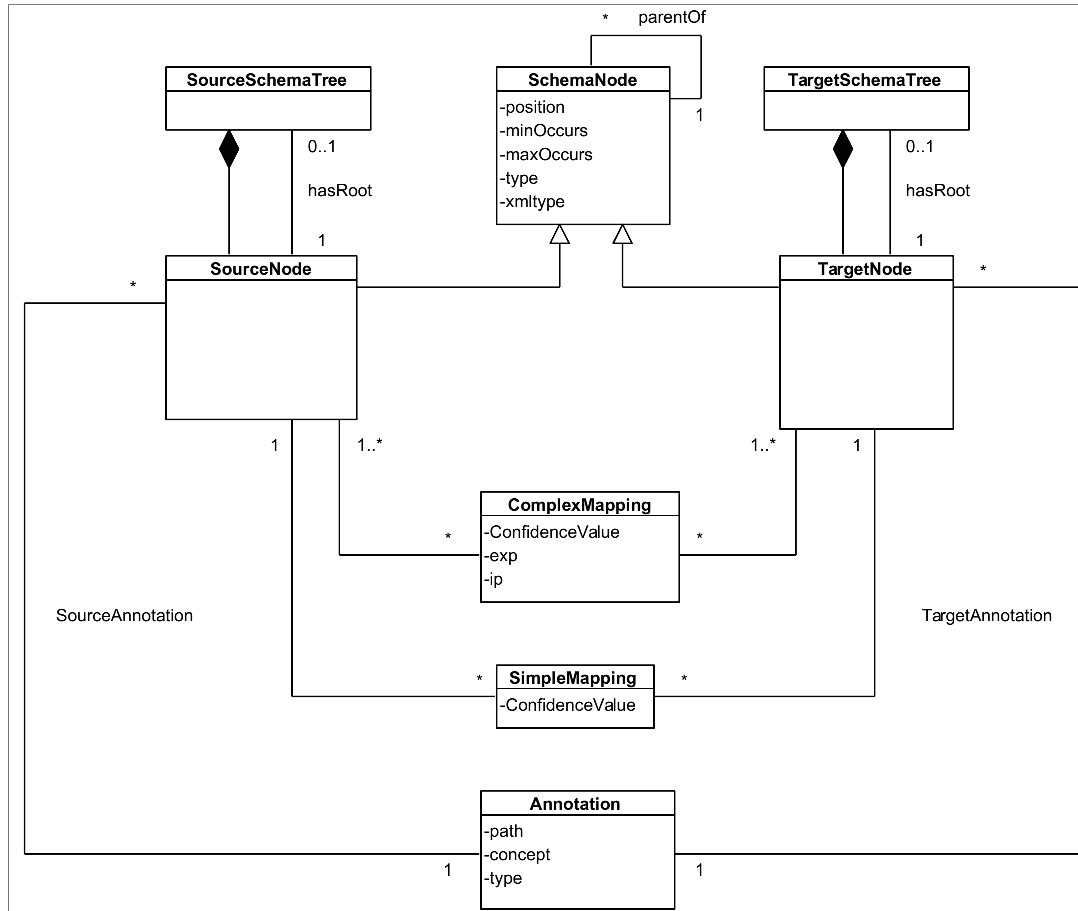

Figure 3.2: A Meta-Model of the proposed XML-Schema mapping approach

the complex mapping relation. A meta-model of the proposed schema mapping formalism is depicted in figure 3.2.

3.2.2 Semantic Relations between Annotations for Simple Matches

In [21] local *1:1* matches are realized by multiple matching algorithms, where each algorithm fills a $|S| \times |T|$ matrix with confidence values. We will now show how such a matcher that exploits the semantic relations between the annotations can be realized using our annotation method. Thus, obviously a node s from the source schema and a node t from the target schema will get a maximum confidence value when the annotation concept of the annotation paths $s.a$ and $t.a$ are equivalent according to the reference ontology. If they are not equivalent but the concept of the annotation of the source node is a subclass of the concept of the annotation of the target node also a high confidence value can be returned. Because semantically any instance of a subconcept is also an instance of the superconcept. However, in this case it is

likely that multiple subconcepts are matched with a common superconcept. It is therefore, desirable to use the closest subconcept for the match. The closest subconcept is defined by the hierarchal distance between the source and target annotation concept. While a match from a subconcept to a superconcept is always possible this does not hold for the opposite direction. Nevertheless, a match from a super-concept to a subconcept can also be correct but this depends on the matching situation. Thus, the confidence level of a superconcept to subconcept match must be very low because it requires user-intervention for the mapping generation. Another type of such a non-strict match is the match between concepts that have the same parent. For example there is a source annotation */BusinessAddress/hasStreet* and a target annotation */PrivateAddress/hasStreet/*, *BusinessAddress* and *PrivateAddress* are subconcepts of *address*. This means they do not match perfectly but as long as *PrivateAddress* and *BusinessAddress* are not defined to be disjoint they are at least semantically related. Therefore, in some situations this match can be a valid match.

After the basic semantic relations between annotation paths are introduced, we can define how those relations can be used for a matcher that operates on the semantic annotations. The general idea is to produce a $n \times m$ matrix filled with confidence values as used in [21]. This allows to combine a matcher that operates purely on the semantic annotations with additional matchers that operate on the structure or on constraints. In order to produce the $n \times m$ matrix we use a semantic matcher. The matcher gets a source node s and a target node t and the enhanced reference ontology O' as input and returns a value between zero for no correspondence and one for equivalence. The matcher is defined as:

$$sm(s,t,O') = \begin{cases} 1 * \alpha & \text{if } e = semEqual(s.a, t.a, O') \\ 1 * \beta * / \frac{1}{ConceptDistance(s.a,t.a,O')} & \text{else if } isSubConcept(s.a, t.a, O') \\ 1 * \gamma * / \frac{1}{ConceptDistance(s.a,t.a,O')} & \text{else if } isSubConcept(t.a, s.a, O') \\ 1 * \delta * / \frac{1}{ConceptDistance(s.a,t.a,O')} & \text{else if } hasSameParent(s.a, t.a, O') \\ 0 & \text{else} \end{cases}$$

The enhanced reference ontology O' is the reference ontology where all annotation concepts from the source and the target schema are included according to the description in section 2.3. The helper function $semEqual(s.a, t.a, O')$ gets two annotation paths and the reference ontology as input and returns *true*, if the annotation concepts are equivalent, otherwise $false$. The function $isSubConcept(s.a, t.a, O')$ returns *true* if the source annotation concept is an equivalent- or subconcept of the target-annotation concept. The parameters α, β, γ, δ are used to define the maximum confidence level that the specific type of match can return. The helper function $hasSameParent(s.a, t.a, O')$ returns true, if the annotation concept of $s.a$ and $t.a$ have the same parent (except *Thing*) and are not defined to be disjoint. $ConceptDistance(s.a, t.a, O)$ returns the hierarchical distance between the concept of $s.a$ and $t.a$ in the *isA* hierarchy of O'. It is defined as the minimum number of nodes that needs to be traversed in the hierarchy in order to reach the annotations concept of $s.a$ from the annotations concept of $t.a$.

The matching needs to be done separately for concept annotations and datatype annotations, since otherwise concept annotations may be matched with datatype annotations which does

not reflect the intended semantics (see section 2.3). In case of datatype annotations (annotation paths that end with a datatype-property) the method *semEqual*() must only return *true* if the datatype properties are also equivalent. The methods *isSubConcept*() and *hasSameParent*() must only return *true*, if there is also a corresponding sub-property relation between the datatype properties.
The matching function operates only on the semantic annotation. In order to achieve good match results it needs to be combined with an additional matcher that operates on the constraints of the XML-Schemas. Therefore, we propose to use the given semantic matcher as one matcher of a composite matcher.
Since the given matcher operates only on the semantic annotations the construction of the $n \times m$ matrix with confidence values does actually not need $n \times m$ calls to the matcher. It is sufficient to only relate each source annotation to each target annotation because multiple schema elements can have the same annotation. In order to disambiguate matches between multiple nodes with the same annotation the additional structural matcher can help.

3.2.3 Semantic Relations between Annotations for Complex Matches

In the last section we have presented an approach for the generation of 1:1 matches. In practical scenarios it turned out, that they are already useful for the majority of cases. One advantage of such matches is that they are supported by generic matching systems such as [21] or [2]. Therefore, such systems can easily be extended with a semantic matcher that operates on the proposed annotation method.
We have shown in section 3.1.2 that all existing systems that allow the creation of complex matches need to have extra knowledge. The approaches achieve this either by correlation mining on multiple schemas, by data-mining over instance data or by predefined domain- or XML-snippets that express the relations between different XML-elements.
Since we are interested in a purely schema level approach the evaluation of instance data is not an option. Instead, we want to use the annotations to find suitable complex matches. If the ontology could define how different ontology elements can be transformed, then the source of knowledge could be the ontology itself. Unfortunately, the OWL reference ontology in our case is limited to description logic reasoning over instances and classes. Complex matching expressions or templates can therefore, not be defined on the OWL level. In Semantic Web applications those transformations can be realized with rule languages such as SWRL [78]. Unfortunately, those rules operate on instance data of the ontology. In our schema-level approach this does not help because instance data is never lifted to the ontology. Therefore, we propose to solve this problem by using transformation templates that contain the required knowledge. Those transformation templates have a set of inputs and a set of outputs. Each element of the inputs and outputs is annotated with the reference ontology.

Definition 3.2.1. *Transformation Template* A transformation template t is a tuple $t = (S, T, exp, IC, CM, ip)$, where S is a set of inputs, T is a set of outputs, each element $e \in S \cup T$ has an annotation $e.a$. The attribute exp is an arbitrary expression that transforms the inputs to the outputs. IC is a set of XML-level integrity constraints that needs to be met in order to instantiate the template, CM is set of tuples, that define the required context mapping for inputs and outputs. ip is a boolean value that indicates if the transformation expression is information preserving or not. In general a transformation expression exp is information preserving, if there can exist another expression that allows a mapping in the opposite direction.

The goal of transformation templates is to create complex matching elements:

Definition 3.2.2. *Complex Matching Element* A complex matching element is a tuple (SM, TM, exp, ip, cv), where SM is a set of source mappings, TM is a set of target mappings, exp is an arbitrary expression that transforms the input to the output. Each element in SM is a tuple (s, i), where s is an element of the source schema and i is an input of exp. Each element in TM is a tuple (o, t), where t is an element of the target schema and o is an output of exp. The value cv defines the confidence value (see subsection 3.2.3) of the matching element. When ip is set to true, then exp is information preserving.

In order to show the requirements for a complex match between a set of nodes from the source schema and a set of nodes from the target schema that instantiates some transformation template, we will first introduce the decomposition of an annotation path into its postfix and context. An example is shown in figure 3.3.

Definition 3.2.3. *Postfix and Context of two annotation paths* $p1$, $p2$. Given an annotation path $p1$ that consists of n steps and an annotation path $p2$ that consists of m steps, such that $n > m$. The postfix of $p1$ with regard to $p2$ ($postfix(p1, p2)$) is a sub-path of $p1$ that consists of the rightmost m steps of $p1$. The context of $p1$ with regard to $p2$ ($context(p1, p2)$) is a sub-path of $p1$ that consists of the leftmost $n - m$ steps of $p1$. If the context ends with an object-property, then the missing element is replaced with the top-concept *owl:Thing*.

After the general terms have been introduced, we can discuss the required relations between the source annotations of a schema, a transformation template and the annotations of the target schema in order to establish a complex matching element. We will first introduce the problem with an example shown graphically in figure 3.3. Given a source node s with the annotation *s.a=/Order/has/ItemList/contains/Item/has/DollarPrice* and a target node t with the annotation *t.a=/Document/has/ItemList/contains/Item/has/EuroPrice* and a transformation template $t = (\{in\}, \{out\}, exp, \{\}, \{\}, true)$, where *in* is annotated with *in.a=DollarPrice* and *out* has the annotation *out.a=EuroPrice*.

We can show that $s.a \sqsubseteq in.a$. Therefore, $s.a$ can provide data with the correct semantics for the input of the transformation template. This also means that we can create a new path *p1=Context(s.a,in.a) + out.a* with the help of the transformation template. When this path can

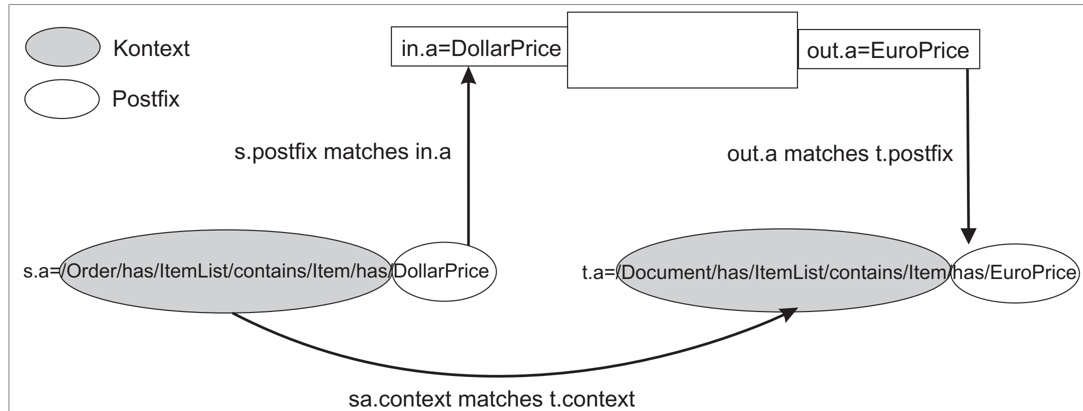

Figure 3.3: Finding complex matches with annotated transformation templates

be matched with *t.a*, then a match with the template can be established. In the example this is obviously the case because *p1=/Order/has/ItemList/contains/Item/has/EuroPrice* is a subclass of *t.a=/Document/has/ItemList/contains/Item/has/EuroPrice*.

When we assume that we do not want to match one source node with one target node but a set of source nodes S with a set of target nodes T and a set of transformation templates $TEMPL$, we would end up producing new paths for each node $\in S$, whenever there is a match with an input of a template. Those new paths would then need to be matched with all nodes $\in T$. This gets unfeasible because creating new paths results in high computational costs for the reasoner. Therefore, we will inspect the required relations in more detail in order to decompose the problem into sub-problems that can be solved more efficiently.

We have depicted the general idea using the previous example in figure 3.3. The idea is that instead of creating new paths for each output of a matching template we only need to proof that: $Context(s.a, in.a) \sqsubseteq Context(t.a, o.a) \wedge Postfix(s.a, in.a) \sqsubseteq in.a \wedge out.a \sqsubseteq Postfix(t.a, out.a)$.

This has the advantage that $Context(a, i)$ and $Postfix(a, i)$ are only defined on the length of i. Thus, we only need to create additional concepts for each different length of input for the source schema and for each different size of output for each target schema. Because practically many elements of a document have the same context and inputs of transformation templates typically have a length of 1 to 2. The additional effort of creating new annotation concepts is strongly limited. Another advantage is that all those additional paths can be created in advance. Thus, the reasoner can classify them in a preprocessing step.

We will now propose and proof that it is sufficient to match the contexts and the source postfix with the input and the output with the target postfix.

The function $length(path)$ returns the number of steps of $path$.
$Replace(p1, in, out)$ replaces the postfix of length $length(in)$ with out. The resulting path has the length $length(p1) - length(in) + length(out)$.

Theorem 3.2.1. *Given the annotation paths* $p1$*,* $p2$*,* in*,* out*:*
$p1 \sqsubseteq in \wedge Replace(p1, in, out) \sqsubseteq p2 \wedge Replace(p1, in, out) \sqsubseteq Thing$[1]
$\Leftrightarrow$
$Postfix(p1, in) \sqsubseteq in \wedge$
$out \sqsubseteq Postfix(p2, out) \wedge$
$Context(p1, in) \sqsubseteq Context(p2, out) \wedge$
$Replace(p1, in, out) \sqsubseteq Thing$

Proof. $\Rightarrow$
From $p1 \sqsubseteq in$ directly follows that $Postfix(p1, in) \sqsubseteq in$. The reason is that a longer annotation path a always defines more specialized concepts than a shorter one b. $Postfix(a, b)$ has the same size as b. If $Postfix(a, b) \not\sqsubseteq b$, then a cannot get a subconcept of b by getting more specialized. Both concepts get unrelated.
From $Replace(p1, in, out) \sqsubseteq p2$ follows, that $Postfix(Replace(p1, in, out), out) \sqsubseteq Postfix(p2, out)$. But $Postfix(Replace(p1, in, out), out)$ is equivalent to out, therefore, $out \sqsubseteq Postfix(p2, out)$.
Finally from $Postfix(Replace(p1, in, out), out) \sqsubseteq Postfix(p2, out) \wedge Replace(p1, in, out) \sqsubseteq p2$ follows that $Context(p1, in) \sqsubseteq Context(p2, out)$ The reason is that two annotation path a, b, where $a \sqsubseteq b$ get hierarchically incomparable, when they are combined with contexts, where $a.context \not\sqsubseteq b.context$.

$\Leftarrow$
From $Postfix(p1, in) \sqsubseteq in$ directly follows $p1 \sqsubseteq in$.
From $out \sqsubseteq Postfix(p2, out)$ follows that $Postfix(Replace(p1, in, out), out) \sqsubseteq Postfix(p2, out)$.
From $Context(p1, in) \sqsubseteq Context(p2, out)$ follows that $Context(Replace(p1, in, out), out) \sqsubseteq Context(p2, out)$ because $Context(Replace(p1, in, out), out) \Leftrightarrow Context(p1, in)$.
When $Postfix(Replace(p1, in, out), out) \sqsubseteq Postfix(p2, out) \wedge Context(Replace(p1, in, out), out) \sqsubseteq Context(p2, out)$, then also $Replace(p1, in, out) \sqsubseteq p2$. □

[1]This ensures that the replaced annotation path is logically valid (see section 6.2.1)

We now use 3.2.1 to specify, when a match between a source and a target schema can be established with a *1:1* transformation template:

Definition 3.2.4. *1:1 Matching-Template* Given a source node *s* with the annotation *s.a* and a target node *t* with an annotation *t.a*, and a *1:1* transformation template $templ = (\{i\}, \{o\}, exp, ICS, \{\}, true)$, a matching element based on *templ* can be created if:

1. $Postfix(s.a, i.a) \sqsubseteq in.a$
2. $o.a \sqsubseteq Postfix(t.a, o.a)$
3. $Context(s.a, in.a) \sqsubseteq Context(t.a, out.a)$
4. $Replace(s.a, i.a, o.a) \sqsubseteq Thing$
5. *No integrity constraint* $\in$ *ICS is violated by the source and target nodes.*

Description: Requirement one to four are a direct consequence of theorem 3.2.1. The additional check of XML-Level integrity constraints ensures that the transformation template does not only fit semantically but also structurally.

Example: Given a source node *s* with the annotation *s.a=/order/hasItem/Item/hasPrice/EuroPrice* and a target node *t* with the annotation *t.a=/document/hasItem/Item/hasPrice/Price/DollarPrice* and a transformation template $t = (\{i\}, \{o\}, exp, \{\}, \{\}, true)$, where $i.a = /EuroPrice$ and $o.a = /DollarPrice$, and en empty set of integrity constraints. The expression exp can for example be defined as *{o=i*1.35}*
We can establish a match between *s* and *t* with the help of the transformation template because *Postfix(s.a,i.a) = EuroPrice* is equivalent to *i.a* and the output of the transformation template *o.a* is equivalent to *Postfix(t.a,o.a)=DollarPrice*. Finally the context of the annotations match because */order/hasItem/Item/hasPrice/Thing* is a subclass of */document/hasItem/Item/hasPrice/Thing*.

After we have shown, when a match with a *1:1* template can be established, we will generalize the match requirements for *n:m* transformation templates. In case of *n:m* transformation templates, there is no direct mapping between the input and output nodes. Therefore, the required mapping relation for the contexts is defined by the set of context-mapping tuples CM, where each tuple $\in CM$ has the form (in, out), where *in* references an input element and *out* references an output element.

Definition 3.2.5. *n:m Matching-Template* Given a set of source nodes SN and a set of target nodes TN and a *n:m* transformation template $templ = (IN, OUT, exp, ICS, CM, true)$, where IN is a set of annotated inputs and OUT is a set of annotated outputs of the transformation expression exp. CM is a non empty set of context mappings. A match between a subset of SN and a subset of TN with the template *templ* exists, if:

1. $\forall\ i \in IN\ \exists\ s \in SN \mid Postfix(s.a, i.a) \sqsubseteq i.a$

2. $\exists\ o \in OUT \wedge \exists\ t \in TN \mid o.a \sqsubseteq Postfix(t.a, o.a)$

3. $\forall\ cm \in CM\ \exists\ (s \in SN \wedge t \in TN) \mid Postfix(s.a, cm.in.a) \sqsubseteq cm.in.a\ \wedge$
   $cm.out.a \sqsubseteq Postfix(t.a, cm.out.a)$
   $\Rightarrow Context(s.a, cm.in.a) \sqsubseteq Context(t.a, cm.out.a) \wedge Replace(s.a, cm.in.a, cm.out.a) \sqsubseteq Thing$

4. No integrity constraint $\in$ *ICS* is violated by the source and target nodes.

Description: A *n:m* transformation template can only be used, if all preconditions are met. Those preconditions are, that all inputs can semantically be matched with the postfix of annotations of source nodes. In addition, at least one output must be matched with the postfix of a target node. This is a direct consequence of theorem 3.2.1. The only additional requirement is that the defined concept mappings are not violated. This requirement is directly inline with the definition of the context mapping.

Example: Given a source schema with the element *nettoPrice* that represents the total net-price and the element *tax* that represents the tax-ratio in percent. The target schema has one element *bruttoprice* that represents the total-price including the tax-value and an additional element *includedtax* that represents the tax-amount that is included in the *bruttorpice* element. The corresponding annotations are: *nettoPrice.a=/order/hasTotalPrice/nettoPrice*, *tax.a=/order/hasTaxRatio/Taxratio*, *bruttoPrice.a=/order/hasTotalPrice/bruttoPrice*, and *includedtax.a=/order/hasTotalPrice/includedTaxAmount*.

We now assume to have a transformation template *t=({i1,i2},{o1,o2},exp,{},{(i1,o1),(i2,o2)},true)*, where *i1.a=nettoPrice*, *i2.a=Taxratio*, *o1.a=bruttoPrice*, and *o2.a=includedTaxAmount*. The expression *exp* can be an expression like: *{o1=i1*(1+(i2/100)); o2=i1*(1+(i2/100))-i1}*. We can establish a match because all inputs of the transformation template can be matched with inputs of the source schema: *Postfix(nettoPrice.a,in1)* $\sqsubseteq i1.a$ and *Postfix(tax.a,i2.a)* $\sqsubseteq i2.a$. In addition all outputs of the transformation template can be matched with nodes from the target schema: $o1.a$ $\sqsubseteq$ *Postfix(bruttoPrice.a,o1.a)* and $o2.a \sqsubseteq$ *Postfix(includedtax.a,o2.a)*. Finally each tuple in the context map *{(i1,o1),(i1,o2)}* can be matched: *Context(nettoPrice.a,i1.a)* $\sqsubseteq$ *Context(bruttoPrice.a,o1.a)* and *Context(nettoPrice.a,i1.a)* $\sqsubseteq$ *Context(includedtax.a,o2.a)* because */order/hasTotalPrice/Thing* is equivalent to */order/hasTotalPrice/Thing*.

A Note on Datatype Annotations

A datatype annotation is an annotation that ends with a datatype-property. The annotation method requires that an annotation path must always start with a concept. This has consequences for general transformation templates that define transformations regardless of the context. For example in order to define a transformation template *Euro2Dollar* that transforms data between the datatype properties *hasEuroValue* and *hasDollarvalue* the input and output needs to be annotated with *Thing/hasEuroValue* and *Thing/hasDollarValue*. However, in order to achieve meaningful postfixes and prefixes the *Thing*-step needs to be removed and the

matching of postfix and input and postfix and output must be based on sub-property relations rather than on subclass relations. For datatype annotations that do not start with *Thing*, the sub-property relation needs to be checked additionally (see simple matches in section 3.2.2).

Confidence Values of Complex Matches

After the applicability of transformation templates is defined, we can match nodes of a target schema with complex matches if the described required conditions for the instantiation of a template are met. The matching criteria from definition 3.2.5 ensures that all found complex matching elements are able to create some desired output.

In order to weight the found complex matches according to their match quality a confidence value analogue to the simple 1:1 matches is required. In general confidence values can be interpreted as the probability that some element from the source schema s matches some element t of the target schema. When a direct match is established the probability is the direct result of the semantic annotation based matching function (see section 3.2.2). When a complex *1:1* match is established we therefore, propose to calculate the overall confidence value *matchval* of the complex match using a transformation template *templ* as:

$$\begin{aligned}&matchval = match(postfix(s1.a, templ.in.a), in.a) * \\ &match(templ.out.a, postfix(t.a, out.a)) * \\ &match(context(s1.a, in.a), context(t.a, out.a)))\end{aligned}$$

In case of *1:n*, *n:1* or *n:m* complex matches there is not only one such match but there are multiple matches for each context mapping tuple. In this case the overall value of a complex matching element can only be an aggregation (such as average) of the single match values.

3.2.4 Mapping Workflow

The goal of the matching of the source and target schema is the creation of a directed schema mapping from the source schema to the target schema. The meta-model for the mapping formalism is shown in figure 3.2. A schema mapping is basically a set of simple and complex mapping elements. A simple mapping element relates one node of the expanded source schema tree to one node of the expanded target schema tree. The semantics of such an element is that data from an element of the source document can directly be copied to a an element of a target document. Each node of the source and target schema can participate in multiple simple mapping elements as long as the *min-* and *max occurs* restrictions of the nodes are not violated (global *n:m* cardinality). A complex schema mapping element relates a set of source nodes to a set of target nodes with some matching expression.

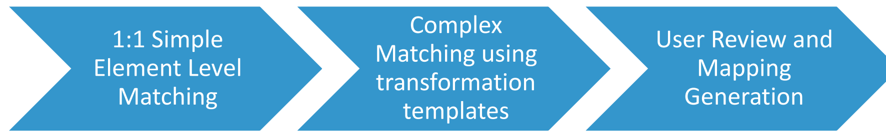

Figure 3.4: Schema mapping workflow

In order to create a schema mapping we propose to use the mapping workflow which is depicted in figure 3.4.

As already discussed in section 3.1.2 the size of the search space for schema matching with complex matches gets unfeasible for or an exhaustive search. Therefore, we use two methods to reduce the search-space. In a first phase *1:1* matches are generated. In the second phase complex matches are only created for not yet matched target elements. The generation of such complex matches is directly based on the semantic relations between the source and target nodes and the existing transformation templates (see section 3.2.3).

In the first phase simple correspondences between schema elements are created using a composite matcher that consists of a semantic annotation based matcher as proposed in subsection 3.2.2 and an XML-Schema constraint based matcher. The output of this phase is a $|S| \times |T|$ matrix that relates each source node to each target node. In order to generate a schema mapping we have the following problem:

Given a $|S| \times |T|$ matrix that includes the computed confidence values and a set of nodes S and T, where each node $s \in S$ or $\in T$ has a *min-* and *max occurs* restriction. We search for a set M of mapping elements (s, t, cv), where $s \in S$, $t \in T$ and cv is the confidence value of s and t from the confidence matrix. The goal is to find a mapping M, where the sum of all confidence values of the mapping elements is maximized, and no constraints over the *min-* and *max-occurs* restrictions of the schema nodes are violated.

In the second phase complex matches for target elements that are not yet targets of *1:1* mapping elements or that only participate in mapping elements with confidence values under some threshold are generated. This can result in multiple complex matches for one specific set of target nodes. We propose to only use that complex matching element with a maximum combined confidence level. This is only one possible strategy - other strategies could for example be to (additionally) minimize the number of template instantiations and to maximize the number of matched target nodes. All those strategies require experiments on annotated real-life schemas and data-sets for their evaluation, which is not in the scope of this research. The user can also define new templates if a match could not be established with the existing templates from a template library. Such a user-defined template can later automatically be reused because the inputs and outputs are annotated. Finally a schema mapping candidate can be generated by adding the complex matches to the set of already generated mapping

elements. This mapping candidate can then be presented to the user who can still make modifications.

3.3 Mismatch Resolution

The main goal of the matching and mapping approach is to provide interoperability between different XML-Schemas. We will now first present an example for a schema mapping that was established using the proposed annotation and mapping method and then discuss how different types of mismatches [69] can be resolved.

We suppose to have a business case, where it is required to create a directed mapping in order to transform order documents from a company that only sells furniture to private customers to a schema for order documents of some mail-order house that sends any kinds of goods to any kind of customer. Both partners have agreed on a common business ontology (depicted in figure 3.5) that describes their domain. In addition both schemas are annotated using the proposed annotation method. The schemas are heterogenous. The source schema provides extra XML-elements for the customer- and for the company- address, while the target schema provides only one element *InvolvedParties* that contains sub-elements for the address of the buyer and seller. The source document always uses *EURO* as currency, while the target schema only uses prices in *USD*. Last but not least both schemas use a totally different structure for the guarantee information. In the source schema the type of guarantee is provided by an attribute, while in the target schema there is an own schema element for each different type of guarantee. The automatically generated mapping between both schemas is shown in figure 3.6. The complete source and target schemas including all annotations can be found in the appendix 1, 2, and 3. In the next subsections we will discuss, how the different elements can be mapped.

The work in [69] provides types of mismatches that typically occur between heterogenous schemas. We will discuss the mismatch resolution based on this classification. The proposed mismatch types are divided into lossless and lossy mismatches.

3.3.1 Lossless Mismatches

Lossless mismatches are mismatches that can be revolved without loosing information. As a consequence a bidirectional information preserving mapping is possible.

Naming

The general problem of a naming mismatch is that elements with the same semantics use different labels in the source and target schema. In the example schemas, an example for a

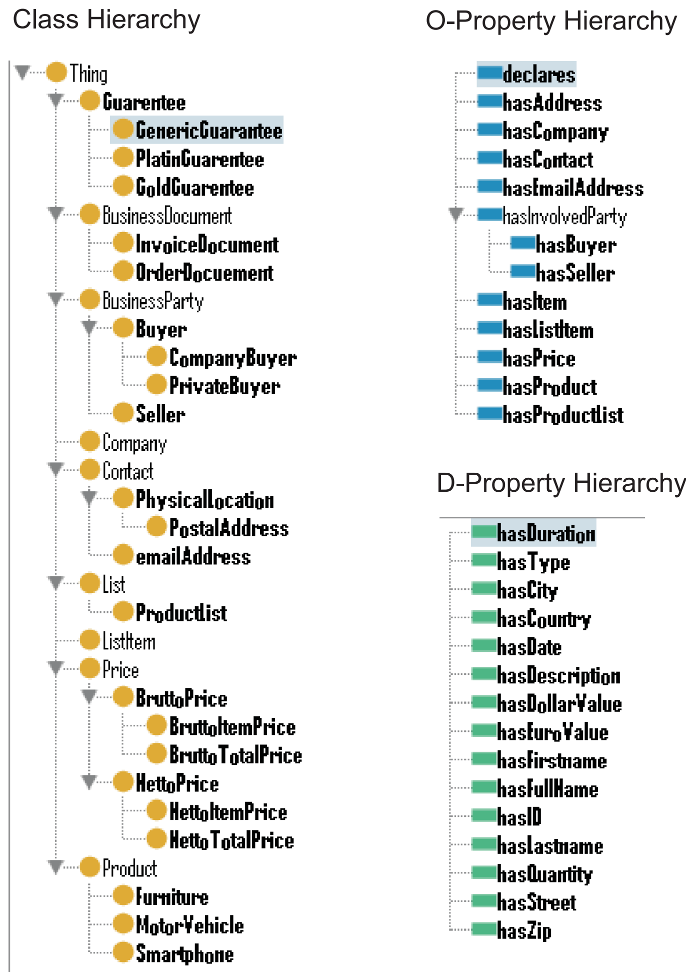

Figure 3.5: Example reference ontology

naming-mismatch are the elements *SellerCompany* and *SellerCompanyName*. The naming mismatches can easily be resolved using the proposed annotation and mapping method. The reason is that the matching is based on the annotations. A mapping can be established if the annotation concept of the source element is a sub- or equivalent concept of the target annotation concept. In the case of the *seller-company* this is trivially the case because the annotation strings are equivalent. Both use the annotation */InvoiceDocument/hasSeller/Seller/hasCompany/-Company*.

Structure Organization

In general a structural organization mismatch means that the same content is structured differently. An example for this are the different representations of the address-data of *buyer* and

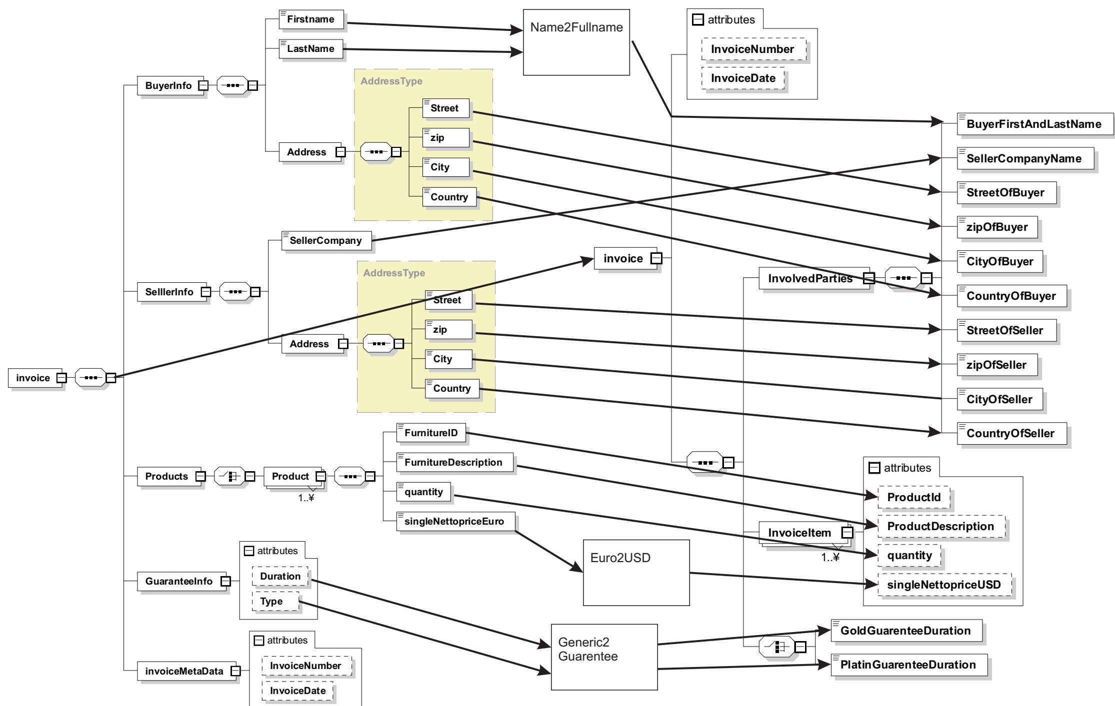

Figure 3.6: Example mapping

seller. The source schema uses two different elements *BuyerInfo* and *SellerInfo*, with nested elements for the corresponding attributes, while the target schema has a flat form with a sequence of elements.

Nevertheless, *1:1* matches between the elements can be achieved by using the semantic annotations. In this case the annotation path for the *buyer* elements are not equivalent (using *privateBuyer* instead of *Buyer* for the annotation) in the source and target schema. A directed mapping is still possible because the annotation concepts of the source schema are subconcepts of the annotation concepts of the target schema. Therefore, the example shows a structure organization mismatch that occurs together with an abstraction mismatch.

Attribute Granularity

In this case the same information is represented by a different number of attributes in the source and target schema. An example is the representation of *FirstName* and *LastName* in the source schema and the usage of one combined element *BuyerFirstAndLastName* in the target schema. The reference ontology in the example provides a specific property *hasFullname* and the distinct properties *hasFirstname* and *hasLastname*. As a consequence the elements are annotated differently. A one to one match is not possible. In order to solve this issue a transformation template needs to exist in the transformation template library. In our case, we

assume there is a transformation template of the form: *FirstnameLastname2Fullname = ({in1, in2 }, out1, exp, {}, true)*, with the annotations *in1.a=/Thing/hasFirstname*, *in2.a/Thing/hasLastname* and *out1.a=Thing/Fullname*. In order to establish a match using this transformation template we need to show that a match based on the annotation of the source and target schema can be established. The source schema contains the elements *FirstName* with the annotation */InvoiceDocument/hasBuyer/PrivateBuyer/hasFirstname/* and *LastName* with the annotation */InvoiceDocument/hasBuyer/PrivateBuyer/hasLastName*. The target schema contains the element *BuyerFirstAndLastName* with the annotation */InvoiceDocument/hasBuyer/Buyer/hasFullName/*. Since the context */InvoiceDocument/hasBuyer/PrivateBuyer* matches */InvoiceDocument/hasBuyer/Buyer/* (subclass match) and the prefixes matches as well, a complex matching element of the form $(\{(s1, in1), (s2, in2)\}, \{(out1, t1)\}, exp, true, cv)$ can be created.

Subclass-Attribute and Schema-Instance

In case of a *Subclass-Attribute* mismatch in the one schema a generic class is used with a specific attribute that defines the actual type, while in the other schema specific types are used. In case of the schema instance problem, data holds schema information (for example expressed as an enumeration of values in the one schema that corresponds to different types in the other schema) in general.

Our proposed approach operates only on the schema level and for this kind of problem the actual instance data needs to be evaluated. Only transformation expressions can have access to instance data. In the example source schema, there is an element *GuaranteeInfo* with the attributes *Duration* and *Type*. The type can either be *Gold* or *Platin*. In the target schema two different elements (*GoldGuarenteeDuration* and *PlatinGuarenteeDuration*) are used to express how long the specific guarantee type is granted. As a consequence different annotations are used in the source and target schema. In the source schema the attribute *Duration* is annotated with */InvoiceDocument/declares/GenericGuarantee/hasDuration* and the attribute *type* is annotated with */InvoiceDocument/declares/GenericGuarantee/hasType*. In the target schema *GoldGuarenteeDuration* is annotated with */InvoiceDocument/declares/GoldGuarentee/hasDuration* and the element *PlatinGuarenteeDuration* has the annotation */InvoiceDocument/declares/PlatinGuarentee/hasDuration*.
Obviously those elements do not match directly. Instead, a transformation template of the form *GenericToSpecificGuarentee = ({in1, in2 }, out1, exp, {}, true)*, where *in1.a=GenericGuarantee/hasDuration* and *in2.a=GenericGuarantee/hasType* and *out1.a=/GoldGuarentee/hasDuration* and *out2.1=/PlatinGuarentee/hasDuration* exists. This template allows the creation of a complex mapping element.

Encoding

In this case different format of data or unit of measure are used in the source and target schema. We propose that elements with different encoding have non (strongly) matching annotations. Thus, a mapping is not directly possible. The resolution lies in the automatic match with transformation templates that define how the mapping can be realized. In the example the price is denoted in *Euro* in the source schema and denoted in *USD* in the target schema. A transformation template of the form *Euro2USD = ({in1,}, out1, exp, {}, true)*, where *in1.a=Thing/hasEuroValue*, *out1.a=Thing/hasUSDValue* is used to solve this issue.

3.3.2 Lossy Mismatches

In case of lossy mismatches it is not possible to generate a bidirectional information preserving mapping.

Content

Different content is denoted by the same concept. This mismatch can be detected because in this case both elements must have different annotations with regard to the reference ontology. A mapping can be generated if a corresponding transformation template exists.

Coverage

The absence of information. The annotations clarify the semantics of the schema elements. This also addresses the detection of non mappable elements due to missing elements.

Precision

Precision mismatches allow either no mapping or if a transformation template can be found only the production of a directed mapping from the schema with more precise elements to the schema with less precise elements.

Abstraction

Level of specialization refinement of the information. This can be detected with the help of the annotations. It typically results in unidirectional mappings. In the example we could match a *private-buyer* with a *buyer*. The opposite direction is not (always) possible. Another example for an abstraction mismatch are aggregations. In this case a transformation template can help

to aggregate the values. This also results in a directed mapping. We will provide an example for such a template.

Given the template *ItemsToTotalPrice = ({Itemlist, /ItemList/has/Item/hasPrice}, TotalPrice, exp, IC, false)*. The template is stated to be capable to calculate the $TotalPrice$, given an $Itemlist$ and the prices of items. There is a set of integrity constraints for that expression and the expression is stated to be not information preserving (it cannot be reverted). When we assume to have two schema elements $s1$ and $s2$ of a source schema, where *s1.a=Order/has/Itemlist*, *s2.a=Order/has/Itemlist/hasItem/hasPrice* and an annotation *t.a=Order/has/TotalPrice* of some element t of a target schema, we can generate a complex mapping element of the form $(\{(s1, in_1), (s2, in_2)\}, \{(out_1, t1)\}, exp, false, cv)$ based on the transformation template.

3.3.3 Discussion

The proposed annotation and mapping method can be used to establish simple and complex mappings between different schemas. The annotation path expression do not need to match syntactically - instead, a semantic match that uses the class and property hierarchy of the reference ontology is used. The structure and naming of elements does not matter to find simple matches based on the annotations.
In order to allow complex mappings a library of transformation templates is required that defines how different concepts can be matched using a complex matching expression. Those transformation templates can be defined generally (in the example the *Euro2Dollar* transformation template can transform *Euro* to $Dollar$ in all contexts) or including a context. The resolution of some mismatches such as schema instance and sub-class attribute mismatches needs to deal with instance data. The only elements that can have access to instance data in our schema level approach are the XML-based transformation templates. This has the drawback that in order to create annotations for those elements the reference ontology needs to contain concepts for the different representations, (for example the *GenericGuarantee* in the example ontology) and transformation templates must be created that allow transformations between the different representations.
In a lifting/lowering approach the addition of such concepts to the reference ontology is not required since the lifting scripts/rules can encapsulate the knowledge and actively transform the instance data to a common representation. An alternative annotation approach that adds runtime knowledge to the annotations is used in [98]. In this case the annotations provide declarative knowledge on how the data can be transformed to instances of the reference ontology at runtime. While the direct application of this idea is not an option in our case, this idea can potentially be used to automatically generate transformation templates and additional concepts by using such enhanced annotations.

3.4 Proof of Concept Implementation

As described in section 3.2 our annotation method can be used to enhance traditional matching approaches with the semantics of schema elements. Preliminary tests have shown that we can even achieve good matching results when the semantic matcher is used as the only matching criteria. Therefore, we have implemented a prototype that mainly operates on the semantic similarity. Additional similarity measures are only incorporated to distinguish ambiguous annotations. Our implementation operates in three phases. In the first phase the semantic annotations are matched. This is an optimization step that avoids matching the same annotations twice. In a second phase the results from the annotation mapping phase is used to map the nodes of the source and target schemas. Finally the output is generated. The prototype is implemented using Java in combination with the OWL API[2] and the Pellet[3] OWL-DL reasoner. We will briefly describe the implementation here and refer the interested reader to [96] and [95] for details. The implementation operates in three phases as shown in figure 3.7. We will describe the phases in the next subsections.

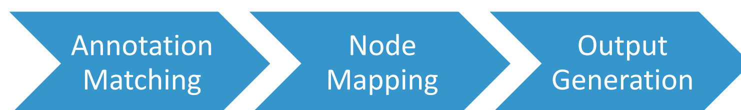

Figure 3.7: Phases of the proof of concept implementation

3.4.1 Annotation Matching Phase

In the annotation matching phase the source and target schemas are first converted to expanded schema trees. An expanded schema tree is a schema where every element that references another type or element is expanded with the definition of the referenced type/element unless no further expansion is possible. The annotations are rewritten according to the rules in section 2.2.2. Finally all annotations are extracted from the expanded source and target schema trees. The annotations are then transformed to ontology concepts as described in section 2.3 and are then added to the reference ontology that we then call the extended reference ontology. In a next step the semantic similarity between each source and target annotation is computed as described in section 3.2.2. This step only produces local *1:1* matches without a matching expression. Therefore, in a next step the annotations of all elements of a library of transformation templates are used to create matches for the not yet matched target annotations.
In contrast to the proposed general transformation templates that allow *n:m* mappings the mapping in the implementation is limited to local *1:1* and local *n:1* mappings because local *n:m* and local *1:n* are not supported by Altova MapForce[4] which is later used to process the

[2]http://owlapi.sourceforge.net/
[3]http://clarkparsia.com/pellet/
[4]http://www.altova.com/de/mapforce.html

mappings. However, this limitation can practically be solved by the usage of multiple *1:n* mappings. The prototype can create matches with existing transformation templates. Such templates can directly be used to create as a matching element or they can be combined to produce a composite matching element. A composite matching element is created by composing multiple transformation templates in order to fulfill some transformation requirement. Such composite matching elements are generated automatically during the matching task by using a simple iterative deepening search algorithm. Finally the user can define additional matching templates that are automatically reused for future matches. By now XML-Level integrity constraints of transformation templates are ignored. Finally the result of the annotation matching phase is a set of simple and complex mapping elements.

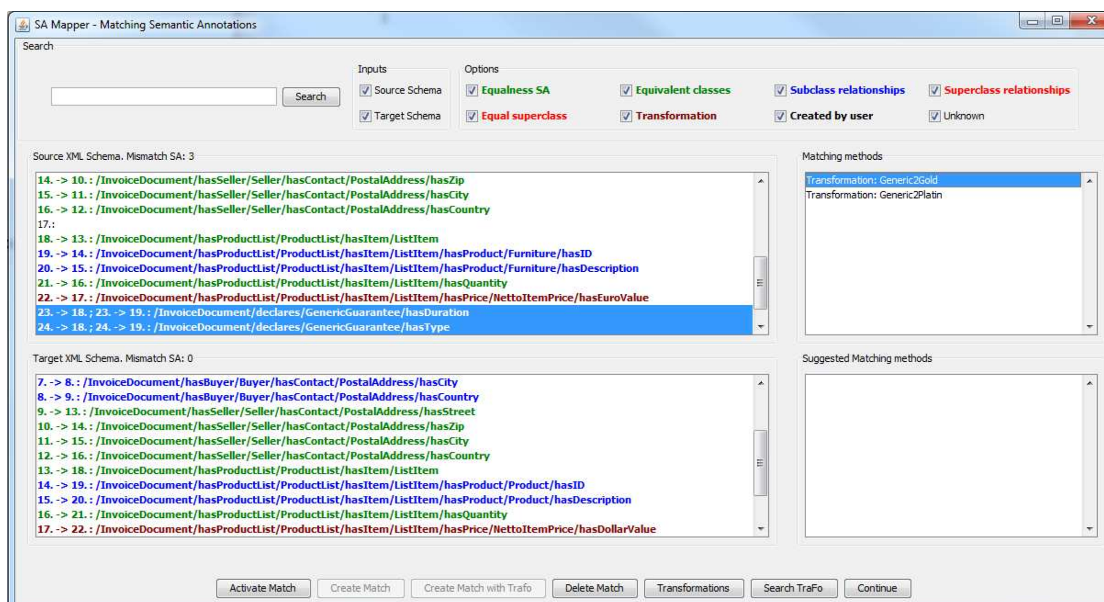

Figure 3.8: Screenshot of the semantic matching phase of the prototype

A screen-shot of the optional user interaction for the mapping of the example schemas from section 3.3 is shown in figure 3.8. Depending on the type of semantic match the matches are shown in different colors. We use green for equivalent matches, blue for subclass to superclass matches, red-brown for matches using transformation templates, and red for super-class matches. In this phase the user has also the chance to modify the mapping and to create additional transformation templates. When a new template is created the annotation matching phase is restarted in order to generate a matching that includes the new transformation template.

3.4.2 Node Mapping Phase

After the possible matches for the annotations haven been discovered they are used to map the nodes of the expanded schema trees. Given an expanded schema tree S of the source schema and an expanded schema tree T of the target schema and a set of mappings M we search for a mapping of nodes MN that map each annotated node $\in T$ to an annotated node $\in S$. For each mapping $m \in M$ there exists a set of nodes $\in S$ where $node.annotation = m.sourceAnnotation$ called *sourceCandidates* of m and a set of nodes $\in T$ where $node.annotation = m.targetAnnotation$ called *targetCandidates*. Depending on the sets *sourceCandidates* and *targetCandidates* there are different cardinalities for the mapping of nodes:

- 1:1: $|targetCandidates| = 1$ and $|sourceCandidates| = 1$
- 1:n: $|targetCandidates| = 1$ and $|sourceCandidates| > 1$
- n:1: $|targetCandidates| > 1$ and $|sourceCandidates| = 1$
- n:m $|targetCandidates| > 1$ and $|sourceCandidates| > 1$

1:1 Mapping:

The *1:1* mapping is the most common case because a schema typically does not contain different nodes with the same semantics. In this case the mapping of nodes is straight forward: Given an annotation mapping $M\ (sourceAnnotation, targetAnnotation)$ with a *1:1* relation between *sourceCandidates* and *targetCandidates*. *SourceNode* is the one and only node $\in$ *sourceCandidates* and *targetNode* is the one and only node $\in$ *targetCandidates*. Thus, a node mapping n = (*sourceNode*,*targetNode*) is created.

1:n and n:1 Mapping:

Nevertheless, there are situations where other cardinalities can occur. Since the semantics of the nodes in non *1:1* relations are basically equivalent, other information needs to be used to find a suitable mapping of the nodes. The relevant information from the XML-Schema that we use are: Cardinality restrictions and the distance between the nodes in the source and target schemas.
One common cause for a *1:n* mapping are different cardinality restrictions on XML-elements in the source and target schema. In the *1:n* case there is one single *sourceNode* $\in$ *sourceCandidates* and there is a set of target-nodes in *targetCandidates*. The cardinality of the source node *sourceNode.cardinality* is bigger or equal to the sum of the cardinality restriction of each *targetNode* $\in$ *targetCandidates*. In this case we can map the first n values of the *sourceNode* to the first *targetNode* which has the max cardinality n. The next m values of the *sourceNode* can be mapped to the second *targetNode* which has the max cardinality m. This can be repeated

until all (potential) values are distributed. Of course this is only a heuristical solution which will produce a valid mapping but user-intervention is required to confirm this solution since also any other mapping would semantically also be possible. In the case of a *n:1* mapping the opposite scenario happens and distinct elements are mapped to one repeatable element.

n:m Mapping:

In case of *n:m* relations both *sourceCandidates* and *targetCandidates* are sets with more than one element. To find node mappings we exploit the location of the nodes in the source and target schema with the assumption that nodes that are semantically related are typically grouped together in a schema. Therefore, we use the following mapping strategy for *n:m* relations:
A node $sn \in sourceCandidates$ is mapped to a node $tn \in targetCandidates$ if the nearest neighbor node of sn, called nns, that is already mapped to a node $tnd \in T$, and the target node tn has the minimum distance to tnd in T. The minimum distance is defined as the number of XML-nodes that need to be traversed along the shortest path between two nodes in the expanded schema tree.

3.4.3 Output Generation

Our proof of concept implementation exports the discovered mappings in form of Altova MapForce[5] project files. The user can therefore, review and - if required - change the discovered mapping with the graphical MapForce tool. Finally an XSLT Script can be generated using MapForce. This Script can then be used to transform instance documents.
A screen-shot of the fully-automatically generated output of the mapping example from section 3.3 using our prototype is shown in figure 3.9. In contrast to the mapping from figure 3.6 the generated mapping uses two different transformations to generate the guarantee elements in the target schema. Therefore, it uses a global *1:n* cardinality to simulate the local *1:n* cardinality which is not supported by MapForce.

3.5 Performance Evaluation

In the previous sections we have shown how the proposed declarative semantic annotations can be used to generate mappings between different XML-Schemas. We have also discussed how mismatches can be resolved using the annotations and transformation templates and have presented a prototype implementation of a schema mapping solution that operates on the proposed declarative semantic annotations. A complementary approach for semantic annotation based document transformation is to annotate the schemas with references to lifting scripts,

[5] http://www.altova.com/de/mapforce.html

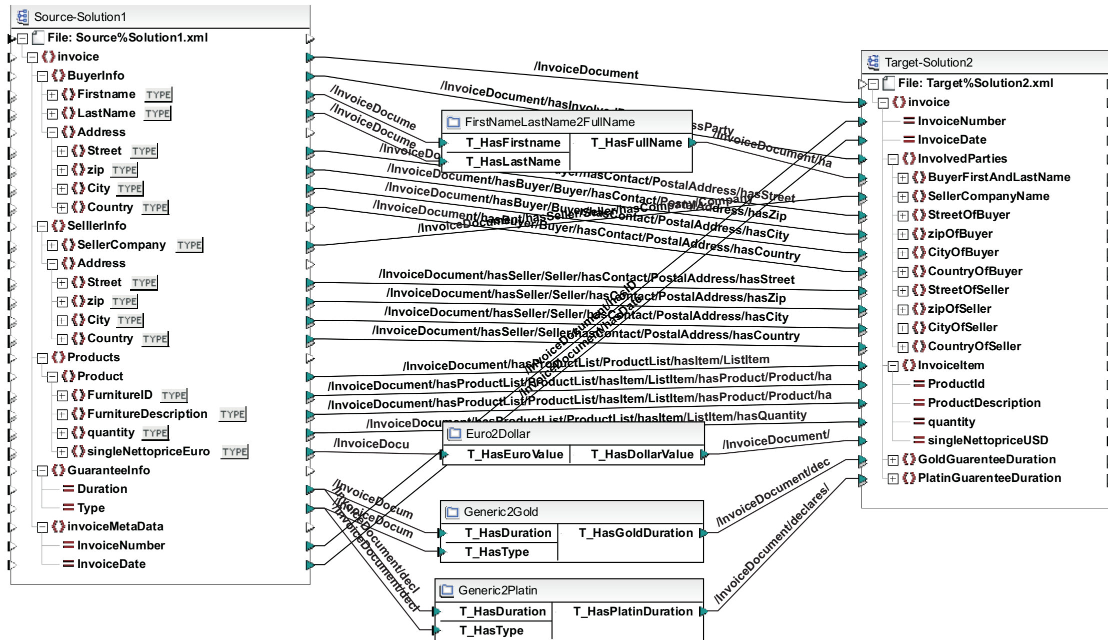

Figure 3.9: Generated output opened in Altova MapForce

that transform instance data to ontology instances (lifting) and back to XML-Documents (lowering). In this section we will evaluate the performance of our prototype implementation against a typical lifting and lowering approach. We will first describe the implementation of the lifting/lowering approach and then provide benchmark results for both approaches.

3.5.1 Lifting/Lowering Implementation

As proposed in SAWSDL an XML-Schema can be annotated with lifting and lowering scripts. Document transformations can be realized by using the lifting scripts that are referenced from the source schema to lift the instance data to the ontology and the lowering scripts that are referenced from the target schema to lower the data to the target XML representation. The lifting/lowering approach is depicted in figure 1.1. The lifting of the instance data allows full reasoning over the instance data. This can just be ontological reasoning with a standard reasoner as well as the application of rules. We have implemented this approach using the Jena Framework[6] and the pellet reasoner[7]. Our implementation is a straight forward application of standard Semantic Web technologies and frameworks and basically performs the following steps:

1. Each document is transformed to RDF [66] using a lifting XSLT [15] script.

[6]http://incubator.apache.org/jena/
[7]http://clarkparsia.com/pellet/

2. The RDF data is added to the reference ontology.
3. An OWL-DL reasoner (pellet) is used to classify the reference ontology including the instance data.
4. JENA rules are used to allow additional transformations that are not possible with plain OWL/SWRL.
5. SPARQL [82] queries are used to query the ontology for the output data. The resultset is represented as XML according to the SPARQL Query Results XML Format [3].
6. The lowering XSLT script is used to transform the XML result to the output format.

Obviously the most time-consuming task in this setting is reasoning over the ontology (3) which is applied for each document. In order to speed up this approach we added bulk-processing. This means the reasoner is not started for every single document. Instead, a set of documents is loaded to the ontology and reasoning is only done once for all the documents in the bulk-job. Afterwards all the documents of the bulk-job are lowered to the target files. The optimal size of the bulk-job is a tradeoff of the size of each document and the size of the ontology. Unfortunately, the lowering requires two steps (5-6). XSPARQL[3] was supposed to combine those steps but unfortunately there is currently no sufficient tool-support for it available. At least we could speed up the XSLT processing by instantiating each XSLT script only once.

3.5.2 Evaluation Setting

We have evaluated the different approaches by using two different XML-Schemas for documents that contain offers for goods. We annotated the documents with a reference ontology that is based on the *goodRelations*[8] ontology. The *goodRelations* ontology only covers the relations between business entities. Therefore, we added an additional concept hierarchy that describes the offered goods as well as additional properties for some not covered aspects of the source and target schema. The product hierarchy is used to evaluate the scalability of the approach with different ontology sizes. Therefore, we varied the number of products in the product hierarchy. The source and target annotations vary in a way that subclass and equivalent class relations need to be exploited to match the schemas in the declarative approach as well as to query instance data for lowering in the lifting/lowering approach. In addition there are the following differences: In the source schema *zip-code* and *city* are represented as two elements and in the target schema there is only one element that contains both values. The source schema requires the prices to be given in Euro, while the target schema assumes Dollar values. Finally the source schema does not contain a *totalprice* attribute that is required by the target schema. Therefore, it needs to be calculated by aggregating the prices of all goods

[8] http://www.heppnetz.de/projects/goodrelations/

in a document. In the lifting and lowering approach those mismatches are solved on the ontology level using Jena rules. Another candidate language for those rules is SWRL [78] which is directly integrated into the pellet reasoner. Unfortunately, it does not support aggregation functions. Our declarative annotation based transformation approach solves these mismatches with transformation templates. All details of the evaluation setting and the implementation can be found in the technical report [54].

3.5.3 Experimental Results

We have evaluated the performance of our schema matching based approach against the lifting/lowering implementation with regard to the overall transformation time for a set of documents. In figure 3.10 the results for the transformation of an offer document that contains only one offered item are shown. The x-axis uses a logarithmic scale in order provide a better readability of the diagram. We varied the size of the used reference ontology. The line depicted *Schema-Map* uses our declarative annotation based schema mapping approach with a small reference ontology without additional products. The line *L/L* shows the lifting/lowering approach for that reference ontology. In contrast *Schema-Map-6065* and *L/L-6065* use a larger reference ontology with 6065 additional product categories. Finally *Schema-Map-15665* and *L/L-15665* use a reference ontology with 15665 additional categories.
When only one document is transformed, the *L/L* approach takes 0.2 seconds. In contrast the schema mapping based approach takes 2.6 seconds due to the additional startup-time for the creation of the mapping. When 1000 documents are transformed the *L/L* approach takes 16.8 seconds, while the schema-mapping approach needs only 7 seconds. This is a performance gain for the schema mapping approach of 1:2.4. The break-even of both approaches is reached at approx. 200 documents.
When the ontology size increases as shown in line *Schema-Map-6065* also the startup time for the mapping generation increases to 11 seconds. When 1000 documents are transformed *Schema-Map-6065* takes 15 seconds, while *L/L-6065* takes 35 seconds. Which is 1.84 times the time of the schema-mapping approach. The break-even of both approaches is reached at approx. 350 documents. Finally, when the ontology gets even bigger the start-up time for schema mapping increases to 23 seconds. In order to transform 1000 documents *Schema-Map-15665* needs 27.5 seconds compared to 65.2 seconds for *L/L-15665*. This is a factor of 1:2.3. The break-even of both approaches is reached at around 380 documents.
The line *Rule-Based* shows the results for an approach that transforms the documents based on Jena rules by avoiding the usage of the ontology. This approach provides a better and ontology-size independent performance than the *L/L* approach but does not provide the same scalability as the *Schema-Map* approach.

In figure 3.11 the results for the transformation of documents that contain 10 offered items are shown. The transformation of one document using *L/L* takes 0.3 seconds. In comparison *Schema-Map* takes 2.6 seconds. When 1000 documents are transformed *L/L* takes 103.3 sec-

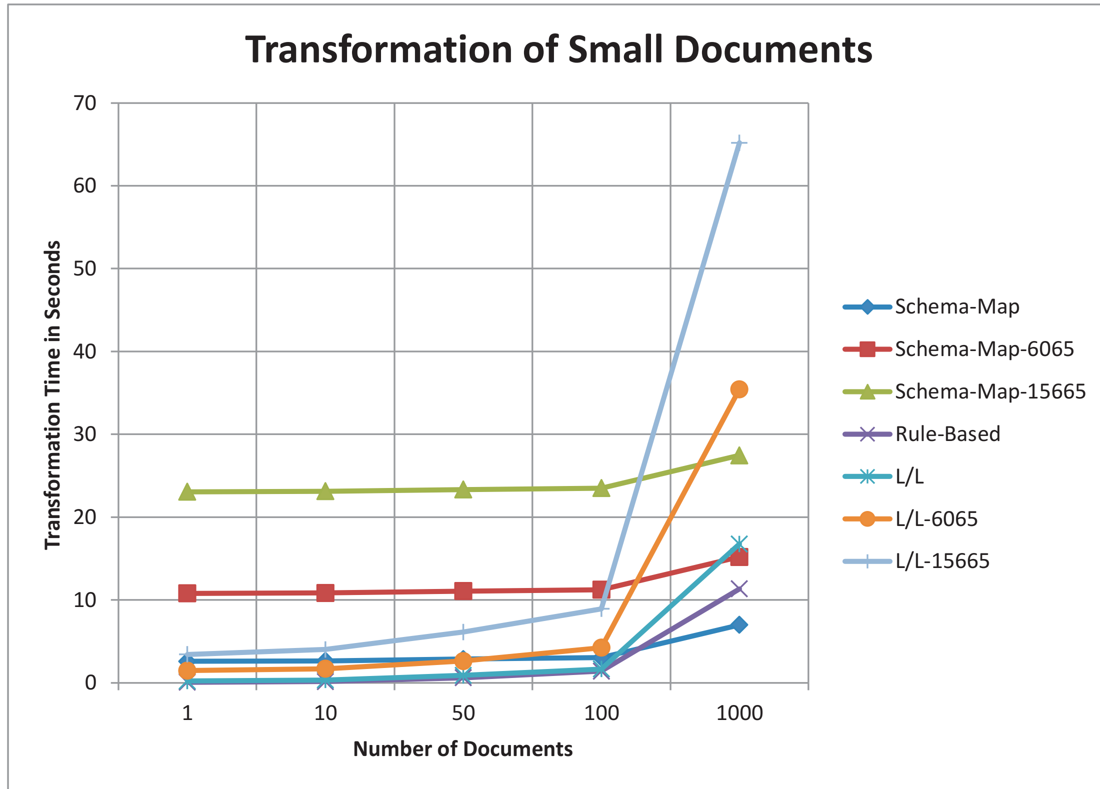

Figure 3.10: Transformation duration in seconds for n documents containing 1 item

onds compared to 9.85 seconds for *Schema-Map*. Thus, the lifting/lowering approach is 10.48 times slower than the schema based approach. The break-even between both approaches is reached at approx. 25 documents. When the ontology size increases *(Schema-Map-6065)* requires 15.16 seconds for 1000 documents, while *L/L-6065* needs 218 seconds. The schema-map approach outperforms the lifting/lowering approach by a factor of 1:14. The break-even of both approaches is reached at approx. 50 documents. Using the big ontology *Schema-Map-15665* requires 30 seconds for 1000 documents, compared to 481.35 for *L/L-15665*. This is 16 times more for the lifting/lowering approach. The break-even is also reached at around 50 documents.
The ontology-size independent pure rule based approach is slightly faster than the *L/L* approach with 75 seconds instead of 103.3 seconds for 1000 documents.

Finally figure 3.12 shows the results for a larger document that contains 100 offered items. In this case *Schema-Map* requires 34.9 seconds for 1000 documents compared to 3820.8 seconds of *L/L*. The schema-mapping approach outperforms the lifting/lowering approach with a factor of 1:109. The break-even is reached before the first document. When the ontology size increases *(Schema-Map-6065)* needs 43.15 seconds compared to 4859.89 seconds of *L/L-6065* to transform 1000 documents. This is 112-times faster, than the lifting/lowering approach. The break-even

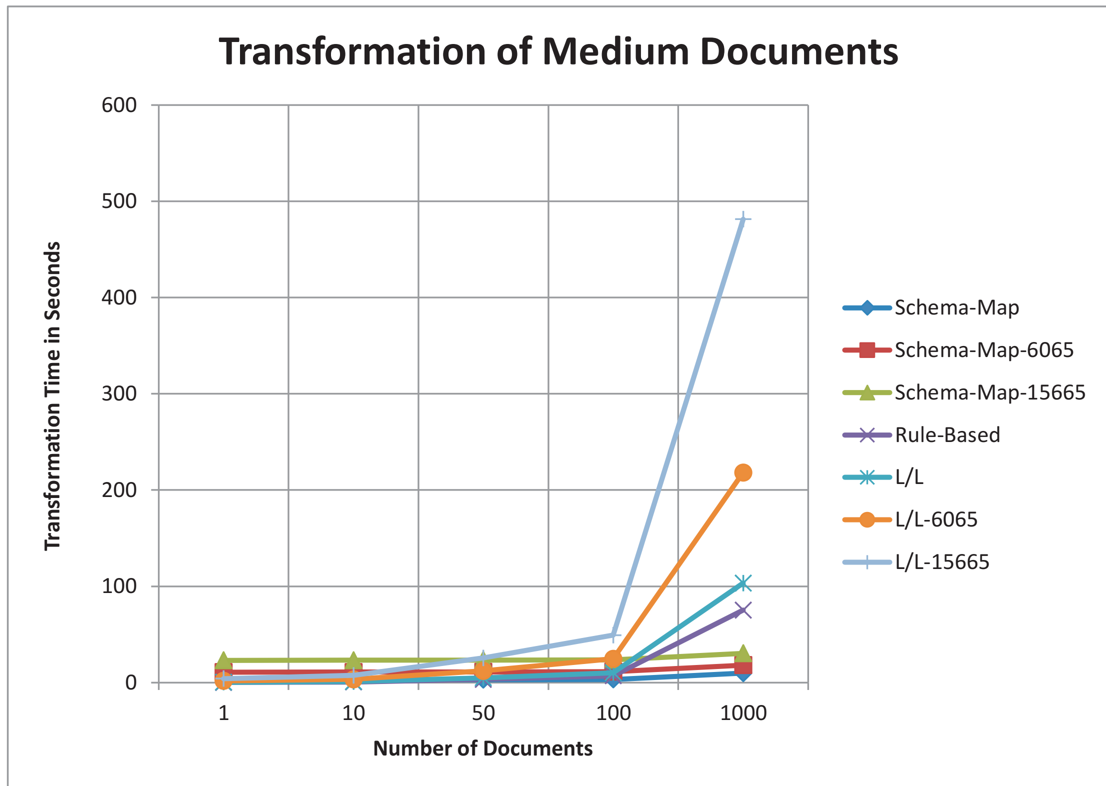

Figure 3.11: Transformation duration in seconds for *n* documents containing 10 items

is reached at around 2 documents. Finally using the big ontology *Schema-Map-15665* takes 55.38 seconds compared to 7203,91 seconds of *L/L-15665*. This is a factor of 1:130. The break-even is reached at approx. 3 documents. The ontology-size independent rule-based approach gets a much worse performance, when the document size increases. It provides about the same performance as the *L/L-15665* approach.

Conclusion: The evaluation clearly shows that the hypothesis of chapter 1 that the proposed build-time creation of XSLT-scripts that solely operate on the XML-level provides a better scalability than a lifting/lowering approach was correct. It provides a much better scalability with regard to the number of transformed documents, ontology size and document size. The mapping generation has the disadvantage of additional startup costs[9]. When very small documents need to be transformed the additional costs are compensated after 200 to 380 document depending on the ontology size. When the document size increases slightly to 10 offered items the break-even point is already reach after 25 to 50 documents. Finally when the document size increases to 100 offered items per document the setup costs loose their impact with a break-even between 0.65 and 3 documents. When it comes to the overall transformation

[9]We used hand-written scripts for the lifting/lowering implementation. When the lifting/lowering mappings were generated automatically this time needs to be added as a startup-cost for the lifting/lowering implementation as well.

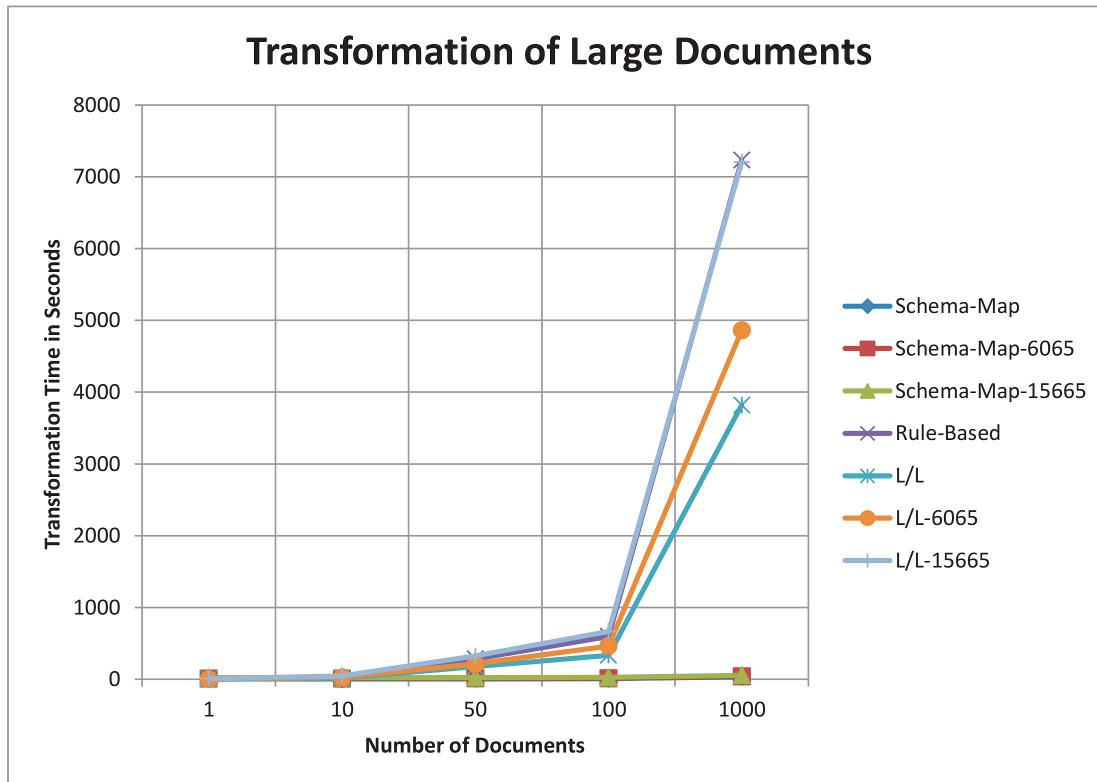

Figure 3.12: Transformation duration in seconds of n documents containing 100 items

time for 1000 documents the schema mapping approach provides a performance advantage between 1:1.84 to 1:1.2.3 for small documents, 1:10 to 1:16 for medium and 1:109 to 1:130 for big documents. This is a tremendous performance advantage for the schema based mapping approach. The additionally tested rule-based approach that does also not require the ontology at runtime was also clearly outperformed - especially, when the document size increases. The results clearly show that the schema-based approach is the right choice for industry-scale transformation scenarios.

3.6 Conclusion

In this chapter we have first introduced the problem of schema matching. Schema matching is used to find correspondences between schemas in order to generate schema mappings. Traditional schema matching approaches try to guess the semantics of the schema elements by exploiting different dimensions such as element or attribute names or the structure of a schema. Those approaches are typically limited to simple correspondences and do not support the generation of complex matches. Complex matches require additional knowledge. Existing

systems either use holistic schema mapping, instance data or explicit additional knowledge to find a limited number of complex matches. In contrast our approach defines the semantics of the schema elements by using declarative semantic annotations. We have therefore shown how semantic annotations can be used to find simple correspondences and complex matches with the help of annotated transformation templates.

We have then shown how different types of mismatches between schemas can be resolved using our annotation and the proposed matching methods. Finally, we have discussed a prototype implementation and have evaluated the performance of our declarative annotation based schema mapping approach compared to a lifting/lowering approach that uses standard Semantic Web technologies and frameworks. The results clearly show that our hypothesis that the schema mapping approach outperforms the lifting/lowering approach for enterprise-scale scenarios was correct.

Chapter 4

Change Representation

We have presented a declarative annotation method for XML-Schemas in chapter 2 and have proposed matching methods for XML-Schemas that are annotated with the proposed annotation method including a prototype implementation in chapter 3. An annotation path basically forms a relation between a schema element and ontology elements. This leads to the problem, that annotations can get invalid due to ontology changes. We have already discussed different kinds of invalidation of annotation path expressions in chapter 1. The different invalidation types are:

- Structural Invalidation
- Logical Invalidation
- Semantic Invalidation (detection of semantic changes)

We will describe the requirements for the change representation for the detection and repair of the different kinds of invalidation in section 4.1. In section 4.4 we will provide a survey of approaches for ontology evolution. As a pre-requirement for the survey we will discuss differences between OWL and frame-based ontologies in section 4.2 and present an example for changes and their consequences for OWL ontologies in section 4.3. Based on the requirements and the survey we will finally describe the change representation approach for this research including its implementation in section 4.5.

4.1 Change Representation Requirements

We need the representation of changes for different tasks: For the repair of structural invalid annotation paths, especially changes of named concepts, properties and instances are of interest. For example the annotation path */inovice/hasBillingAddress/Address/hasZipCode* gets invalid when the last datatype-property *hasZipCode* is renamed to *hasZip* in the new ontology version.

This means we need to know if named concepts or properties were deleted, renamed, split or merged in order to find a suitable repair strategy for the structural maintenance. The logical maintenance of annotation is based on the logical foundation of OWL. In this case it is important to know, which axioms were added or deleted from the ontology in order to explain the logical invalidation. Finally, the detection of semantic changes of annotation paths requires an expressive description of the changes as well as the capability to dynamically query the consequences of changes. Therefore, reasoning support over the old and the new ontology version and the changes is required. We will provide an example for such a query: Has there been an addition of an instance c to the class A or its subclasses, where the added instance c has not been an instance of a subclass of A before? Such reasoning support should be supported by using standard OWL reasoners and frameworks.

This leads to the following requirements for the change representation:

- Structural changes of concepts, properties and instances including composite changes such as rename, split or merge operations.
- A complete axiomatic change-LOG for the justification of logical invalidations.
- Changes should be logged in form of ontology instances that connect the old and the new ontology elements. Thus, reasoning over changes including the old and new ontology can be realized with standard reasoners and frameworks.

4.2 Differences between OWL and Frame-Based Ontologies

The approaches that will be presented in the survey in section 4.4 use different ontology modeling formalisms and languages. The most commonly used ones are frames and description logics (in form of OWL-DL ontologies). We have introduced both briefly in section 1.2.3 of the introduction. Since the different ontology formalisms have a big influence on the representation of changes we will now discuss differences between OWL and frame-based ontologies.

Complex Classes: Anonymous classes that use the different class constructors of OWL allow the definition of arbitrary complex classes. Therefore, compared to frame-based ontologies a class is not organized in form of a static frame with slots and facets - instead it can be constructed by arbitrary logical formulas that uses combinations of class constructors (see 1.2.4). We will provide an example for such a class definition.

$$SingleFather \equiv Person \wedge Male \wedge (\exists\ hasSon.person \vee \exists\ hasDaughter.person) \wedge \neg\ married$$

This is still a comparably simple class expression. An example for deeply nested expression can be found in listing 2.1. Only in simple cases OWL class expressions can be compared

to the slots and facets of frame-based ontologies. In addition there are other fundamental differences between Frames and OWL- ontologies [101]:

No unique name assumption: In contrast to frames OWL does not enforce the unique name assumption. Thus, two things are not considered different because of different names.

Open World assumption: OWL uses the open world assumption. Thus, everything is allowed unless a constraint is definitively violated.

Multiple models: While frame-based ontologies have exactly one minimal model that satisfies all of the assertions there exist multiple models for an OWL ontology: All interpretations that satisfy each of the assertions of the OWL ontology.

Assertion vs. classification: In frames slots and facets of a class define a constraint that all instances of that class must fulfill. While this is also partly supported (with limitation due to the open world assumption) in OWL an additional possibility is provided: Defined classes. They are realized by defining the necessary and sufficient conditions for members of the class. This allows the automatic classification of individuals and subclass-relations based on the constraints and not only based on defined subclass assertions. Therefore, for example a class is logically a subclass of another class if the restrictions are more restricting the possible instances than the restrictions of the superclass.
This is realized by the two axioms of OWL concepts: superclasses and equivalent-classes. Both axioms just link a concept to an anonymous class definition which is basically a logical formula.

Consistency-Checking: In OWL the reasoner that realizes the classification also realizes the consistency-checking by trying to find a model that satisfies all axioms. If this is not possible the ontology is considered to be inconsistent. In Frames the reasoner checks whether all property values on instances fulfill the constraints and no additional types can be assigned to an instance. The open world assumption also has some influence on constraint checking: An instance will not be considered as invalid if it does not have the required properties because the reasoner assumes that it can still have them but they are simply unknown for this instance. An instance or class is only invalid if it cannot get valid with additional knowledge.

OWL-DL which is based on description logics is only one sub-language of OWL. OWL-DL puts some restrictions on the OWL language. For example the set of classes and individuals must be disjoint [1]. Therefore, OWL-DL does not allow meta-modeling and OWL2-DL allows only limited meta-modeling. In contrast OWL-full does not constraint the usage of the

[1]This constraint was relaxed in OWL2 that allows limited [32] meta-modeling support by using punning [99], where the semantics is separated depending on the context.

vocabulary in any way. Therefore, meta-modeling is supported but unfortunately OWL-full is undecidable [70].

4.3 Ontology Changes

The required change operations over ontologies depend on the type of ontology and the used modeling formalism. For example in a taxonomy typical operations are *add*, *delete*, *rename* and changes in the hierarchy as well as the composite *split* and *merge* operations. In case of a frame-based ontology the latter changes are also required but in addition changes to the slots and facets need to be expressed. Finally, in case of OWL ontologies slots and facets can also be expressed but in a different form: A concept is defined via the definition of *EquialentClasses*, *SuperClasses* and *DisjointClasses*. Each of these definitions is expressed via anonymous classes that are just arbitrary logical OWL formulas that may contain restrictions over properties (comparable to slots and facets). This has the consequence that changes do not occur in form of a change of slots and facets but as a change of some axiom. Therefore, a widely used method for change-descriptions of OWL is just a LOG of axioms that were added or removed from the ontology. Unfortunately, in many cases they will not show the real semantics of the change. Therefore, an explicitly stated change may have additional (implicit) consequences for the ontology. This happens in frame-based ontologies as well but in OWL it can also change the class hierarchy due to classification. This leads to the problem that the subclass hierarchy can be changed without ever changing a subclass axiom. We will provide a small example for this behavior:

Class definitions:

```
ClassA
    EquivalentClass(hasB some B and hasC some C)

ClassB
    EquivalentClass(hasC some C and hasD some A)

ClassC
    EquivalentClass(hasA some A and hasB some B)
```

Property Definitions of version 1

```
objectProperty hasA
objectProperty hasB
objectProperty hasC
```

Property Definitions of version 2

```
objectProperty hasA
```

```
objectProperty hasB
objectProperty hasC
    domain: ClassB and ClassA
```

Subclass Hierarchy before the change

```
Thing
    ClassA
    ClassB
    ClassC
```

Subclass Hierarchy after the change

```
Thing
    ClassC
        ClassA == ClassB
        ClassB == ClassA
```

Before the change all classes are on the same hierarchy level. The introduction of a domain axiom on the property *hasC* implicitly changes the class hierarchy because it states that classes that have the *hasC* property are members of the intersection of *ClassA* and *ClassB*. This makes *ClassA* and *ClassB* equivalent in the example. Because the classes are equivalent they both have the properties *hasA* and *hasB* which makes them a subclass of *ClassC*.
The operation *AddAxiom(hasC domain ClassB and ClassC)* resulted in the additional implicit changes: *ClassA equals ClassB*, *ClassB equals ClassA*, *ClassA subclassOf ClassC*, and *ClassB subclassOf ClassC*.

Therefore, changes take place explicitly and implicitly. Depending on the used ontology formalism and type the possible implicit changes can just be the inheritance of additional attributes/slots due to a change in the hierarchy or - as in the case of OWL - they can be arbitrary changes. The problem here is that for the maintenance of annotation and for the detection of semantic changes it makes no difference if the change was made explicitly or implicitly. Any change can have consequences for the annotations. At least it can be assumed that if there was no explicit change than there cannot be an implicit change.

4.4 Survey

In this section we will present approaches for different areas that need to deal with ontology changes. We will present approaches for ontology evolution management, ontology comparison, change-tracing, change modeling, evolution systems, mapping, and multi-version reasoning. The approaches use different kinds of change representation: First, the implicit change representation does not store the actual change, but the state of the ontology elements before

and after the change. Second, the explicit change representation actually expresses what was changed. This explicit change representation can be expressed on different levels of abstraction: Just atomic changes or also composite changes that are composed of atomic changes. Finally, the last representation method are mappings that relate entities from both ontology versions to each other.

4.4.1 Ontology Evolution Management

The idea of ontology evolution management is that an existing ontology needs to be adopted due to some new requirements. Given a consistent ontology we need to apply a set of changes in order to get a new and consistent ontology version that fulfills the new requirements.

User-Driven Ontology Evolution Management

In *User-Driven Ontology Evolution Management* [93] the process of ontology evolution is described. The authors divide the evolution process into 4 phases: Change representation, semantics of change, change implementation, and change propagation.
In the change representation phase the required changes are modeled. The authors use a RDFS-based ontology model that is enhanced with frame-logics, which is described in [94]. The authors state that it is not sufficient to represent changes in an atomic way because this is error-prone and leads to numerous additional changes that are redundant. Instead they propose that the user should define the required changes via composite changes. A list of useful composite changes is provided:

- **Merge Concepts:** Replace several concepts with one and aggregate instances.
- **Extract subconcepts:** Split a concept into several subconcepts and distribute the properties among them.
- **Extract superconcept:** Create a common superconcept for a set of unrelated concepts and transfer common properties to it.
- **Extract related concept:** Extract related information into a new concept and relate it to the original concept.
- **Shallow content copy:** Duplicate a concept with all its properties.
- **Deep content copy:** Recursively apply shallow copy to all subconcepts of a concept.
- **Pull up properties:** Move properties from a subconcept to a superconcept
- **Pull down properties:** Move properties from a superconcept to a subconcept.
- **Move properties:** Move properties from one concept to another concept.
- **Shallow property copy:** Duplicate a property with the same domain and range.

- **Deep property copy:** Recursively apply shallow copy to all subproperties.
- **Move instance:** Moves an instance from one concept to another.

In the second phase, the required atomic changes that are needed for a composite change are computed. There can be multiple possible ways to implement a composite change. The final result can be seen as one specific path in a decision tree. Each branch in the decision tree is called a resolution point and each possible way at a branch is an elementary evolution strategy. Thus, a (non atomic) evolution strategy defines how elementary changes will be resolved. This also includes the resolution of inconsistencies. For example it needs to be defined what to do with the subclasses of a deleted concept. Should they be deleted as well or moved to a superconcept? The idea is that a set of general evolution strategies is provided and the user can select one in order to define how the change should be realized.
In the change implementation phase a list of all changes and implications is computed and shown to the user. The user can approve the change in order to apply them to the ontology. When the changes are done they need to be propagated to the dependent artifacts in the change-propagation phase.

The idea that the actual user-intention is represented using composite changes and that the required atomic changes are computed afterwards, make the strategy very interesting because the intended changes are stored on an appropriate high level in the system.

Consistent Evolution of OWL Ontologies

In [37] consistent ontology evolution is described as the process of creating a new consistent version of a previously consistent ontology. This version is created by adding or removing DL-Axioms. The used ontology model is OWL-DL. The authors address three different types of consistency definitions: Structural consistency (the new version corresponds to the appropriate language class), logical consistency (no contradictions) and user-defined consistency (user-requirements that are expressed outside the ontology), and finally provide ways to ensure a consistent evolution. Methods to find the cause of inconsistencies are described and evolution strategies to resolve them are proposed.

4.4.2 Ontology Comparison Approaches

In this case, given two versions of an ontology the goal is to find the changes that were made in order to transform the old version to the new version. Such approaches are especially of interest if no trace of changes between the ontology versions exist. The approaches are strongly related to schema matching (see chapter 3). The problem of comparing two ontology versions with each other can be generalized to the problem of the comparison of two arbitrary graphs which is known as the graph isomorphism problem. Unfortunately, this problem is

considered to be NP-complete [53]. However, one advantage in case of the comparison of ontology versions is that typically only a very limited set of ontology elements is changed between two versions and that the ontology graph has predefined semantics.

Ontology Versioning and Change Detection on the Web

One of the first approaches for the evolution/versioning of ontologies is described in *Ontology Versioning and Change Detection on the Web* [49]. It is the basis for the ontology evolution tool OntoView [51]. OntoView allows the user to manage different versions of ontologies and to compute the difference between two ontology versions. The differences are computed with a structural-level change-detection approach and are shown to the user by simple highlighting in the graphical user interface. The approach distinguishes the following types of changes: non-logical changes (for example changes of labels), logical definition changes (for example change of subclass statements), identification changes (change of an URI/identifier), addition of definitions and deletion of definitions.
A typical diff[2] approach for text-files is not useful to compare ontologies since syntactical changes do not necessarily result in changes in the ontology model. To overcome this problem not the syntactical representation is compared but its representation in form of RDF-graphs. In a first step all concept definitions of both ontology versions are transformed to RDF-triples. This results in a number of small graphs which define the concepts. Afterwards the small graphs from the source and target ontology are matched by either their URI or their properties. In the last step rules are used to detect changes. A different set of rules is required for each ontology language. The authors also state that depending on the ontology language it can be required to first materialize the ontology-graphs to create a complete concept-hierarchy.
While reviewing the detected changes the user has the ability to specify conceptual relations between the different versions of concepts. The overall approach is inspired by CVS systems. It does not depend on a specific ontology language. The actual implementation supports DAML+OIL [65] and RDF Schema [7]. The implementation does not use a reasoner and can thus, only detect changes that were explicitly stated. The conceptual changes (provided by the user) can be exported in form of a mapping ontology. The mapping ontology imports the source and target ontology and relates the entities with the language constructs than can be expressed with the specific ontology language. The authors state that this is only a partial mapping because not all relations can be expressed.

Prompt-Diff

Prompt-Diff [71] is an adaption of the well known ontology merging algorithm PROMPT [72]. It is based on a fix-point algorithm and a number of heuristic matchers that are used to match two ontology versions. It uses a frame-based ontology model and it was part of the older

[2] Comparable to the diff command on unix-based systems.

versions of the ontology management tool protege. The discovered changes are stored in the so-called prompt-Diff Table. The table consists of tuples of the form:

$< F1, F2, rename_value, operation_value, mapping_level >$

- F1 and F2 are the names of two frames that match.
- rename_value is a boolean that indicates whether a rename occurred $false$ means rename.
- operation_value defines the operation that took place: *add, delete, split, merge, map (none of the others).*
- mapping_level can be $unchanged$, $isomorphic$ (not structurally but logically equal), and $changed$.

Therefore, Prompt-Diff is restricted to the comparison of frame-based ontologies. It supports atomic and some complex change operations.

S-Match

S-Match [79] is a matching algorithm for the matching of specific forms of lightweight ontologies. The input are simple hierarchies of labels. In a first step the node labels are transformed to logical formulas using linguistic processing. For example a node with the name *Herbivore or Carnivore* is translated to the expression $(Herbivore \vee Carnivore)$. If this node is a sub-node of animal the complete formula is $(Animal \wedge (Herbivore \vee Carnivore))$ The result are two lightweight-ontologies [31] which are then used for matching. The result of the S-match algorithm is a matrix that relates every node from the source tree to every node in the target-tree with the strongest detected semantic relation. The possible relations are equivalence, more general, less general, mismatch and overlapping. The algorithm solves the matching problem by proving that the negation of the relation does not hold. Therefore, the matching is resolved with a SAT-solver.

Detection Changes in Ontologies via DAG Comparison

In *Detection Changes in Ontologies via DAG Comparison* [27] an algorithm for the detection of changes between two ontology versions is presented. In order to find the difference between two ontology versions the ontologies are first transformed to rooted, directed acyclic graphs (RDAG). In a next step a change-detection algorithm that is based on [12] computes an edit script that transforms the old ontology version to the new version. The edit script supports the operations: Insert and delete of nodes, edges and slots, update and rename of nodes and update of edge types. The edit script only supports atomic operations and the two complex operations rename and update. The approach operates on frame-based ontologies.

Rules-Based generation of Diff Evolution Mappings between Ontology Versions

The algorithm in *Rules-Based generation of Diff Evolution Mappings between Ontology Versions* [38] starts with a set of basic matches between two ontology versions. Based on these matches the changes that occurred are computed (evolution mapping) and an edit script that transforms the old version to the new version is generated. The evolution mapping does not only contain simple atomic operations but also complex operations. The authors show that the inverse of the evolution mapping always exists and therefore, also a reverse edit-script can be created. The ontology model is based on a Directed Acyclic Graph (DAG). Each node has a unique id and can have concept attributes (like slots in a frame-based system). Relationships (*subClass*, *partOf*, but also arbitrary user-defined relations) between the concepts are expressed in form of arcs. Therefore, the ontology model is a DAG representation of a frame-based system that uses the unique name assumption.
According to the DAG based ontology model change operations to add and delete nodes, arcs and attributes are used. In addition, mapping expression that map a node, arc or attribute from one version to another are provided.
The simple change operations are accomplished with a set of complex change operations. The complex change operations can be applied on single elements such as *substitute*, *move*, *toObsolete*, *revokeObsolete* or can be used on multiple elements: *addLeaf*, *delLeaf*, *Merge*, *Split*, *addSubGraph*, *delSubGraph*. All change-operations are defined in form of rules. Thus, if a rule fires, the specific change operation has occurred.

Logical Difference and Module Extraction with CEX and MEX

The previous approaches typically used structural changes to describe the changes between two ontology versions. In *Logical Difference and Module Extraction with CEX and MEX* [55] an approach is presented that calculates the logical difference between two ontology versions. The approach is limited to terminologies (no multiple inheritance and only the two axioms *equal-Class* and *subClass* are allowed) and does therefore, not support the full expressivity of OWL. The result of the algorithm are logical axioms which, when merged with the old terminology version, result in the new terminology version.

4.4.3 Change-Tracing Approaches

The comparison of two subsequent ontology versions can be avoided if a complete trace of changes between both versions exists. Such a trace can for example be created by an ontology management system. The trace can either consist of the applied changes in form of statements that add, delete or modify ontology elements or by before and after images of the ontology entities. This results in a low-level change representation that makes it hard to figure out the actual user intention of a change. For example a set of atomic changes may actually result

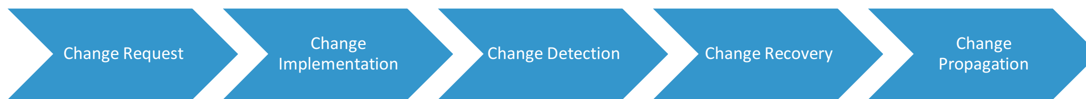

Figure 4.1: 5-phase ontology evolution approach of [80]

in a split. Therefore, the goal of change-tracing approaches is to generate additional change-description that describe the changes on a higher level.

Ontology Change Detection using a Version LOG

The authors of *Ontology Change Detection using a Version Log* [80] argue that an atomic-change-LOG is not useful for other systems and users to actually understand what was changed. Therefore, changes should be described on different levels of granularity.
The main contribution of the paper is a mixed approach that allows the top-down specification of intended changes as well as a bottom up identification of additional complex changes which are detected by evaluating a version log. This leads to the following process for ontology evolution which is depicted in figure 4.1: Change request (basic and composite changes), change implementation (execution of required atomic changes on the ontology), change detection (detection of additional composite changes and meta-changes), change-recovery (maybe the changes induced unwanted side-effects, than the changes can be revised), change propagation (propagation of the changes to dependent artifacts). The version LOG is based on transaction time. Every change of one transaction has the same timestamp and all transactions are ordered along the timeline. The LOG contains versions of ontology entities and not actual change-operations. Based on this version LOG a declarative change definition language is used. The changes are expressed in form of rules that can either be applied on the version LOG in order to generate a change or that can be used to query the version LOG in order to identify changes.

The authors use the typical OWL constructs for their examples but their approach is not limited to OWL ontologies. The general applicability for OWL ontologies is reduced by the fact that all change definitions directly operate on the version log. Thus, if a change occurs that has additional consequences for the classification of some concept in the inference-model this additional change cannot be tracked. An example for such a change is shown in section 4.3. This limits the usefulness of the approach for general DL-based ontologies.

Change Tracer: Tracking changes in Web Ontologies

The authors of *Change Tracer: Tracking changes in Web Ontologies* [47] propose a Change History Ontology (CHO) that logs the changes that happened to an ontology. Changes are described

via change-sets. A change-set has a time-interval, is applied on some resource and contains a number of atomic changes. Atomic changes can be class-, property-, and instance-changes. According to the type of change the following subclasses for each change-subject exist: add, del and update. This results in changes like *ClassAddition*, *PropertyAddition* or *ClassRenaming* or *propertyRangeModification*. The change Ontology is a basis for Change Tracer [46]. No composite changes are supported. The main goal is the ability to roll-back and roll-forward changes in order to retrieve different version of an ontology or to recover to a consistent state.

4.4.4 Change Modeling

The approaches from the last subsections all have their own method to model changes in ontologies. The approaches often rely on a restricted ontology model and the representation methods are mostly not described in detail. In this section we will describe an approach that focuses on change representation.
In *Change representation for OWL2 Ontologies* [76] a complete identification of all possible changes that can be made according to the OWL2 Meta-Model is provided. The result of the work is an ontology of ontology changes. The main classes of this ontology are depicted in figure 4.2. A change-LOG is a set of instances of this change ontology. One key class is the class *changeSpecification* which has an *intitalTimestamp*, a *lastTimestamp*, and a reference to the old and the new ontology version. In addition to this it is associated with a set of changes. The changes are ordered in a sequence via the properties *hasPreviousChange*. In addition each change has an author. The change-class has three subclasses: *Atomic Change*, *Entity Change* and *Composite Change*. An atomic change has the subclasses *add* and *delete* and the property *appliedAxiom*. The range of this property are OWL axioms as defined in the OWL2 meta model. An entity change has the property *relatedEntity* which refers to OWL2 entities from the OWL2 meta model. In order to express what change occurred the *entityChange* class has numerous subclasses:
AnnotationPropertyChange with the subclass *commentchange* and *ClassChange* with the subclasses *SubClassOfChange*, *DisjopintnessChange*, *Class EquaivalenceChange*, ...
The last subclass of change is *CompositeChange* with numerous subclasses for example for *AddSubtree*, *MergeSiblings*, *MoveSubtree*, *SplitClass*. The three subclasses of changes (*Atomic-Change*, *EntityChange* and *Composite Change*) allow the change definition with different levels of granularity. Every change can be represented as a sequence of atomic changes. In order to define that an atomic change belongs to an entity change, entity changes have the property *consistsOfAtomicOperation*. Composite changes have the property *consistsOf* to describe the underlying changes.

Due to the meta-modeling limitations of OWL DL the change ontology cannot directly refer to concepts and axioms of the ontology. In order to overcome this limitation the authors use a meta ontology of OWL that is itself represented in form of an OWL ontology. Thus, the

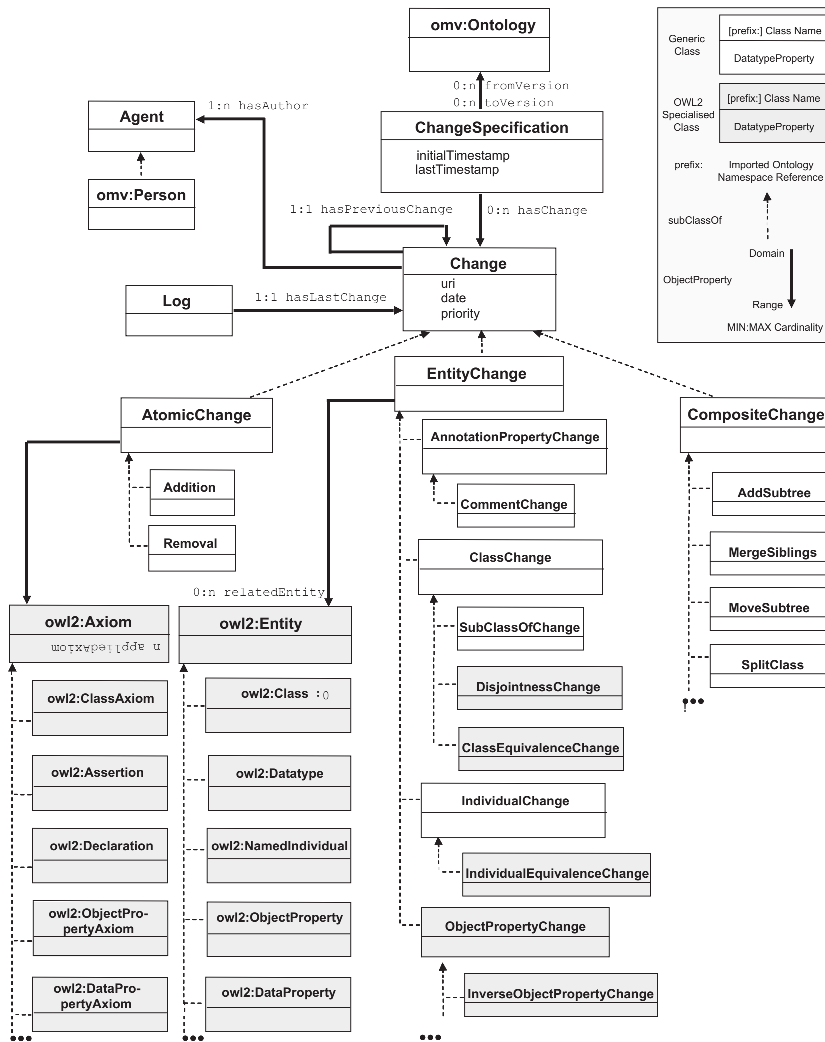

Figure 4.2: The main classes of the OWL2 change ontology [76]

change description can reference entities of this meta ontology.
The approach is used in various plugins of the NeOn Toolkit[3]. The plugins include change capturing, change argumentation and change propagation as well as collaborative ontology development support. Especially the change argumentation plugin "Cicero" is of interest for this survey because it deals with the (manual) definition of changes at higher levels of abstraction. Thus, atomic and entity changes can be annotated with composite changes by the user in order to describe the changes on a higher level.

[3]http://neon-toolkit.org

Modeling Changes in Ontologies

In *Modeling Changes in Ontologies* [25] an ontology versioning approach based on a versioning graph is proposed. The ontology definition is based on classes an relations between classes. A class has slots to assign attributes and properties. Thus, the ontology model is influenced by frame-based systems. Classes and relations both have two timestamps that represent the valid-time of a class or property respectively. In order to manipulate the ontology versioning graph the operations for the insertion, deletion and update of classes and relations are introduced and the effects for the valid-time definition are shown. The granularity of an update operation is the class or relation-level. Thus, an update of a slot is reflected as an update of a concept. The authors present how a specific version of an ontology can be retrieved from the ontology versioning graph and possible implementation approaches are discussed.

4.4.5 Ontology Evolution Systems

The goal of ontology evolution systems it the ability to retrieve different version of one ontology efficiently. These approaches also need to deal with changes that were made in the ontology. These changes can either be modeled in form of before- and after-images, change-logs or time-stamping of each ontology element. Such systems can be a basis for further change-analysis and are therefore, also of interest for the change representation.

A new Approach to Managing the Evolution of OWL Ontologies

A new Approach to Managing the Evolution of OWL Ontologies [13] describes an ontology evolution system that allows the retrieval of a specific ontology version out of a multi-version ontology storage. In contrast to other approaches where ontologies are modeled in form of graphs this approach is solely based on the atomic parts of OWL ontologies: Axioms and annotations. A LOG-file called evolutionary LOG describes the lifetime of every singe axiom or annotation by attaching the two timestamps creation and retirement. Therefore, it is sufficient to check which axioms were created before a given point in time and are not yet retired in order to retrieve some specific ontology version. In addition to the timestamps also meta-data about an axiom such as creator and access information are stored. An axiom or annotation cannot be changed. Only the meta-data can be updated. Thus, if an axiom needs to be changed the original axiom is set to be retired and a new one is introduced. The access to axioms can be restricted and additional meta-data can be provided. The system can thus, be a basis for the collaborative ontology editing. The system allows to retrieve any version of an ontology and implements algorithms to preserve the consistency.
While the approach can be used for any logic based ontology language and all language features are supported, it does not support any detailed information about the semantics of

changes. Compared to other approaches only delete and add operations of axioms and annotations are supported.

Efficient Management of Biomedical Ontology Versions

In *Efficient Management of Biomedical Ontology Versions* [48] a central ontology storage system is described that stores an ontology with multiple versions without redundancies. The key concept of the system is a relational schema that stores ontologies, versions concepts, relationships, relationship type attributes and attribute values. Multiple importers exist for different ontology formats. When a new ontology version is important it is compared with the previous version. For the comparison of the versions a specific property of Life-Science-Ontologies is used: Accession numbers. These numbers uniquely identify a concept and can therefore, dramatically reduce the matching complexity. The matching allows the creation of a list of added and deleted concepts as well as of added and deleted attributes of concepts. The change-set is then used to update the database. Therefore, new concepts or properties are added to the database and deleted concepts or attributes are marked as deleted by adding an end-timestamp.

The applied ontology model assumes the existence of accession numbers and supports concepts and attributes of concepts as well as relationships between concepts. This limits the application to ontologies that have unique ids and do not require more expressive language concepts as found in description logic ontologies. The change-analysis component is only used to import new ontology versions. The changes are not explicitly stored in a change-log.

4.4.6 Ontology Mapping and Multi-Version Reasoning

The evolution of ontologies can also be seen as a special case of an alignment where arbitrary ontologies are aligned. Therefore, we also present an API for ontology mappings. The mapping approaches do not explicitly define what was changed. Instead two ontologies are mapped. The authors of *An API for Ontology Alignment* [29] describe an API that allows the alignment of multiple ontologies in order to generate axioms or rules for transformations. It is based on an abstract alignment format. An alignment is based on a correspondence. A correspondence is a quadruple $< e, e', R, n >$. Where e and e' are entities from the source and target ontology, R is a relation and n is a degree of confidence for the correspondence. The authors describe three levels of alignments:

- Level 0: e and e' are single entities that are addressed by URI's / path expressions and the relation is equal by default but can also be subsumption or incompatibility or even a fuzzy-relation.
- Level 1: Instead of a pair of entities, pairs of sets are aligned.

- Level 2: The mappings are formulated in form of rules including variables.

Once the alignments are defined they can be transformed to different output formats such as OWL axioms or SWRL [78] rules using different renderers. In case of OWL the mappings are transformed in form of bridging axioms that merge both ontologies. The bridging axioms are typical OWL relations such as *subClassOf* or *equivalentClass*. Due to this limitation, this renderer only supports Level 0 and Level 1 mappings.
In contrast, the SWRL rule renderer also supports Level 2 alignments but needs to be adopted to the specific definition language for the level 2 alignments. Since SWRL rules can only operate on instance-data in the ontology this approach is not applicable for our declarative annotation scenario where purely declarative alignments are required because instances are never lifted to the ontology.

The authors of *Reasoning with Multi-version Ontologies: A Temporal Logic Approach* [40] claim that it is not sufficient to analyze the changes on the structural level because dependent application rely on the reasoning result.
Therefore, they have implemented a multi-version reasoning system "MORE" that allows to ask questions on multiple versions in order to detect changes such as: Are all facts from the old version still derivable in the new version? What facts are not derivable anymore? What new facts are derivable from the new version? Which part of the ontology is backwards-compatible and which not? The system is based on the extension of OWL with linear temporal logics, using a linear version space with relative and absolute version numbering, that allows temporal operators such as *AllPriorVersions*, *SomePriorVersion*, *PreviousVersion*, and *Since*. Those extensions can for example be used to query all facts that are not derivable in the new version by simply asking the reasoner for $\neg\, \phi \wedge$ *PreviousVersion* ϕ, where ϕ are the derivable facts. The implementation focuses on queries about the concept-hierarchy but the authors claim that this could easily be extended to other DL-queries. The system does not store the changes explicitly, instead the consequences can be queried. Since the changes are represented implicitly operations like rename or non logical merges cannot be represented/queried. A non logical merge is a merge, where the merge cannot be inferred logically.

4.4.7 Discussion of the Approaches

The presented approaches solve different classes of problems but all include some kind of change representation between ontology versions. The approaches are now compared according to their usefulness for this research. Table 4.1 compares the different approaches with regard to the supported ontology type, their main goal, the used change-format, support for explicit (structural) changes, the support for composite changes and the support for a complete change-LOG for OWL2 ontologies. The dimensions are based on the change representation requirements from section 4.1.
The ontology evolution management approach [93] does not operate on OWL. Nevertheless,

the general idea, that the user describes the intended changes in form of composite change-operations is a valuable idea because it eliminates the problem of the detection of such changes because they are defined explicitly by the user. The work of [37] operates on OWL-DL ontologies and uses an axiomatic change-log. The main goal is to guarantee that a succeeding version is consistent under various aspects (structural, logical and user-defined). This problem is related to the structural, logical and semantic validity of annotations.

All comparison approaches operate on limited ontology models. The approaches [71], [27], [38] use frame-based ontology formalisms, [79] is limited to simple hierarchies of terms and [55] is limited to terminologies. Thus, no ontology comparison approach is suitable for OWL2 ontologies. On the other hand most of the frame-based comparison approaches [71], [27], [38] support explicit changes over the ontology graph. The approach [38] also supports the detection of composite changes. The limited expressivity of the used ontology formalisms of the comparison approaches also results in the fact the the generated changes cannot provide a complete change-LOG that allows to transform one version of an OWL2 ontology to a succeeding version.

The goal of the change-tracing approaches [80] and [46] is to detect high-level changes by evaluating atomic changes of a change-log. However when they are used for OWL they cannot detect high-level changes that only occur in the classified ontology versions because the only operate on the change-log. This limits the usefulness for OWL ontologies.

The ontology evolution systems do naturally not support explicit changes. It is sufficient to store the changes implicitly for this domain. This can be realized by time-stamping of the objects of the frame-based meta-model of [25] and [48] or by times-tamping the axioms of OWL in [13]. We refer to this solution as an implicit change representation. The change-modeling approach of [76] fulfills all the requirements: It is a complete-change-LOG that is based on the OWL2 meta model. It allows atomic and complex changes and it can store the detected changes in form of instances of a change ontology that directly relate meta ontology instances of the old version to meta ontology instances of the new version. However, the approach can express the changes on different levels but it does not support to detect high-level changes. Only a plugin- for manual change-argumentation is mentioned.

The general ontology mapping framework [29] can be used to map arbitrary ontologies. This mapping does not express explicitly what has changed. In order to allow a full-mapping a rule-based level 2 mapping is required, which allows to transform instances of the old version to instances of the new version. Unfortunately such an instance-based transformation is not applicable for our scenario, where instances of the documents are never lifted to the ontology. The MORE system [40] supports OWL and in contrast to all other change representation approaches it allows to query for implicit changes that only occur in the inference model by extending OWL with temporal logics. This makes it very powerful for the detection of the consequences of changes. However, there is no explicit representation of changes and complex operations such as *rename* and non logical *merges* are not supported. This makes the approach well suited for semantic consequence checking but not suitable for the structural repair of annotations.

Ref	Ontology Type	Main Goal	Change-Format	Explicit Changes	Composite Changes	Complete OWL
[93]	RDFS+F-Logic	Evol. Mgmnt.	Graph-Operations	yes	yes	no
[37]	OWL-DL	Evol. Mgmnt.	Axioms Diff	no	no	yes
[51]	RDFS/DAML-OIL	Comparison	Graph diff	yes	yes, manually	no
[71]	Frames	Comparison	Prompt-Diff Table	yes	yes	no
[79]	Light-Weight	Comparison	Logic Relations	no	no	no
[27]	Frames	Comparison	Edit-Script	yes	rename	no
[38]	Frames	Comparison	Edit-Script	yes	yes	no
[55]	Terminology	Comparison	Axiom Diff	no	no	no
[80]	General	Change-Tracing	Version LOG	yes	yes	-
[46]	OWL	Change-Tracing	Change Ontology	yes	yes	-
[76]	OWL2	Change Modeling	Change Ontology	yes	yes	yes
[25]	Frames	Change Modeling	Timestamped DAG	no	no	no
[13]	OWL	Ontology Evol.	Axiom LOG	no	no	yes
[48]	Frames	Ontology Evol.	na	no	no	no
[29]	Generic	Mapping	Mapping	no	no	no
[40]	OWL	Multiversion Reasoning	implicit	no	no	yes

Table 4.1: Comparison of ontology change approaches

[93] User-Driven Ontology Evolution Management
[37] Consistent Evolution of OWL Ontologies
[51] OntoView - [71] Prompt-Diff - [79] S-Match - [27] Detection Changes in Ontologies via DAG Comparison
[38] Rules-Based generation of Diff Evolution Mappings Between Ontology Versions
[55] Logical Difference and Module Extraction with CEX and MEX
[80] Ontology Change Detection using a Version LOG
[46] Change Tracer: Tracking changes in Web Ontologies
[76] Change representation for OWL2 Ontologies
[25] Modeling Changes in Ontologies
[13] A new Approach to Managing the Evolution of OWL Ontologies
[48] Efficient Management of Biomedical Ontology Versions
[29] An API for Ontology Alignment
[40] Reasoning with Multi-version Ontologies: A Temporal Logic Approach

4.5 Change Representation Approach

The discussed approaches use different kinds of ontology models. There are solutions that use a graph-based meta-model for ontologies such as [71], [51], [27], [25], [38]. They are directly applicable for frame-based ontologies. In this case the ontology is treated more or less like a conceptual model. Therefore, operations that modify this model are of interest. In our approach we use the semantic annotations to link schema elements to named elements of the ontology in order to describe them in form of a common conceptual model. Changes of this model require changes of the annotations. Therefore, this kind of change representation is well suited for the structural maintenance of annotations.
In contrast, methods for OWL need to treat changes in form of the addition or removal of axioms. The changes can be expressed implicitly by defining the valid-time of the axioms as proposed in [13] or by using an expressive change description that provides all kind of change-operations over OWL2 like [76]. While this kind of representation can be used to enable ontology evolution and versioning it can still be hard to figure out what actually happened in the conceptual model due to a change. For example the change of a domain or range of a property can change the class hierarchy (see section 4.3). This hierarchy change will never occur in the change-LOG but it will have consequences in the inference model and possibly also for the annotations. In order to get change-operations over the ontology graph - as possible for frame-based ontologies - not the explicit definitions of the ontology versions need to be compared, but the materialized ontologies. Such a comparison can only detect a predefined subset of changes. Especially for the detection of semantic changes of annotations it needs to be possible to query the change representation for additional arbitrary implicit changes. A query could be: Has there been a split of a subconcept of A in the old version such that now all newly created concepts that were related to this split are only subconcepts of B? This kind of queries require reasoning over both ontology versions and the changes. This is partly supported by [40] because it allows reasoning over multiple ontology versions but it does not explicitly store the changes.
Because none of the discussed approaches allows the detection and explicit representation of changes, a complete edit-script and the possibility to reason over multiple versions and the changes, we propose a hybrid approach for this research. It is depicted in figure 4.3.

In our approach the input ontology versions O_n and O_{n+1} are first classified and then transformed to instances of a meta ontology that represents the structure of the named elements of the input ontologies. In a next step the meta ontology instances can be compared using a structural comparison approach such as [38] or [27]. Finally the detected changes are stored in form of instances of a change ontology. These instances directly relate meta ontology instances of O_n to meta ontology instances of O_{n+1}. Since these changes are only structural changes over the meta ontology they do not provide a complete change-LOG that allows to transform the source ontology O_n version to the target ontology version O_{n+1}. Those changes can be stored additionally in form of a simple axiom log. The change representation of our approach

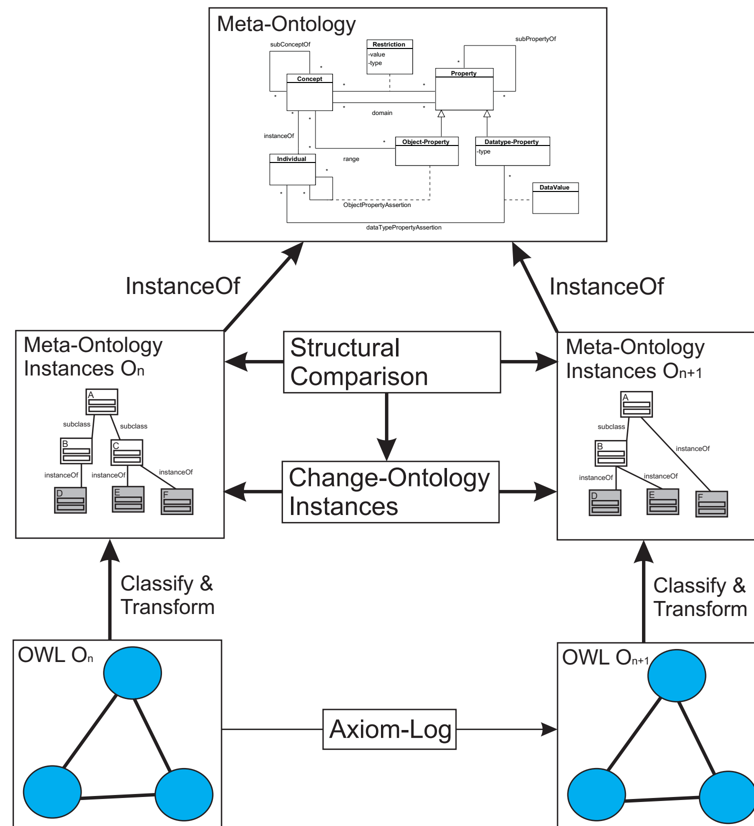

Figure 4.3: Proposed change representation approach

consists of the instances of the change ontology and the instances of the meta ontology. This allows standard reasoning with OWL reasoners and the application of rules and queries on these instances.

4.5.1 Meta Ontology

As already discussed we use a simplified meta ontology that concentrates on the structure of named entities of the ontology. The model is influenced by RDFS and frame-based systems and is depicted in figure 4.4. It contains the relevant aspects of typical ontology languages. Basically an ontology consists of concepts, properties and individuals. Concepts and properties are hierarchically structured. An individual is an instance of a set of concepts. A property has a domain that defines the set of classes that have this property. Properties are divided into object-properties and datatype-properties. Object-properties form relationships between classes and have a range that defines the set of classes which are targets of the property. Datatype-properties have a definition of the data type. Properties are modeled on the class level while an instantiation of a property is done on instance-level using property assertions.

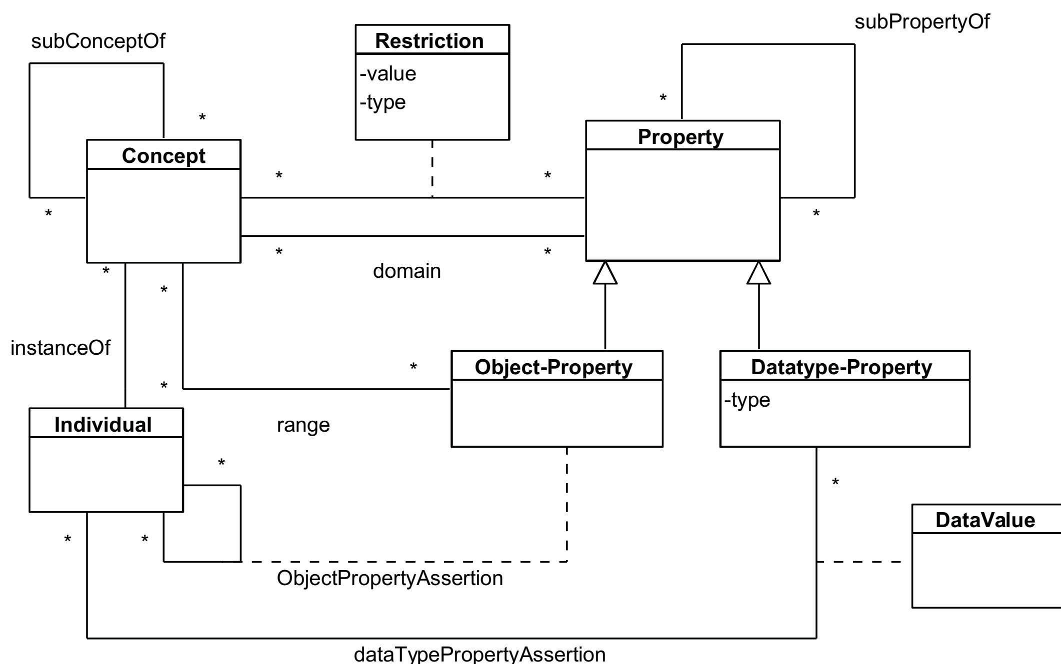

Figure 4.4: Ontology meta-model

Concepts may restrict the usage of properties. A restriction has a type and a value. The type indicates the type of restriction (min, max, value) the value indicates the value that is attached to this restriction. Individuals can have *propertyAssertions* that indicate that an individual instantiates some property. In case of an object-property the target of a *propertyAssertion* is another individual. In case of datatype-properties it is some data-value.

4.5.2 Change Ontology

The change ontology contains a hierarchy of change-operations. The complete list of changes is depicted in figure 4.5. We will now briefly describe the main change-operations and their effects. Each change operations has a unique identifier *tid*.

Global Changes

Global changes add or remove concepts, properties or instances. The table below shows the basic add-operations and their inverse delete-operations.

Figure 4.5: Class hierarchy of the change ontology

c	c^{-1}
addConcept(tid,uri)	delConcept(tid,uri)
addObjectProperty(tid,uri)	delObjectProperty(tid,uri)
addDataTypeProperty(tid,uri)	delDataTypeProperty(tid,uri)
addInstance(tid,uri)	delInstance(tid,uri)

The obvious preconditions of deletes are that the corresponding element must be an element of the corresponding type in the old ontology version. In contrast, the additions requires that a specific element does not exist in the old ontology version. The effect is the existence of the newly added element in the new ontology version.

Hierarchy Changes

In addition to the global operations, hierarchy change-operations are used to express changes in the concept and property hierarchy, as well as of the $instanceOf$ relation of instances. We depict the add-operations and their inverse delete operations in the table below. The abbreviation $cUri$ and $iUri$ stand for the URI of the added or removed child element, $pURI$ for the URI of the parent.

c	c^{-1}
addChildConcept(tid,cUri,pUri)	removeChildConcept(tid,cUri,pUri)
addChildObjectProperty(tid,cUri,pUri)	removeChildObjectProperty(tid,cUri,pUri)
addChildDatatypeProperty(tid,cUri,pUri)	removeChildDatatypeProperty(tid,cUri,pUri)
addInstaceToConcept(tid,iUri,cUri)	removeInstFromConcept(tid,iUri,cUri)

Rename Changes

Rename changes are used to modify the URI of the specific ontology elements. We assume that these operations are global in the sense that every usage of the URI is changed automatically. In addition we assume that these operations are atomic operations. Therefore, a rename does only show up as a rename but not as a delete and a subsequent insert.

c	c^{-1}
renameConcept(tid, oldUri, newUri)	renameConcept(tid, newUri, oldUri)
renameProperty(tid, oldUri, newUri)	renameProperty(tid, newUri, oldUri)
renameIndividual(tid, oldUri, newUri)	renameIndividual(tid, newUri, oldUri)

Update Changes

Update changes are used to modify the domain and ranges of properties as well as to maintain restrictions over properties on concepts and to modify property assertions on individuals. The inverse operations of update changes are update operations of the same type but with swapped parameters.

- updateRestriction(tid, conceptUri, propertyUri, OldValue, OldType, NewValue, NewType)
- updateDomain(tid, propertyUri, {OldConceptUri}, {NewConceptUri})
- updateRange(tid, propertyUri, {OldconceptUri}, {NewconceptUri})
- updateType(tid, propertyUri, OldDataType, NewDataType)
- updatePropertyAssertion(tid, instanceUri, propertyUri, oldvalue, newvalue)

Composite Changes

In addition to the basic operations a set of composite operations is of interest. A merge operation merges a a set of concepts, properties or instances to one single concept, property or instance. The inverse operation of a merge is a split.

c	c^{-1}
mergeConcept(tid,{conceptUri},conceptUri)	splitConcept(tid,conceptUri,{conceptUri})
mergeProperty(tid,{propertyUri},propertyUri)	splitProperty(tid,propertyUri,{propertyUri})
mergeInstance(tid,{instanceUri},instanceUri)	splitInstance(tid,instanceUri,{instanceUri})

A composite operation is reflected as a sequence of other change-operations. In order to specify that an atomic change-operation is part of a composite change the atomic operation is annotated with the *tid* of the corresponding composite change with statements of the form: $ChangeAnnotation(tidOfCompositeChgange, tidOfAtomicChgange)$

Implicit Changes

In addition to provided explicit changes there are implicit changes. These changes can directly be caused by explicit changes. We expect the following implicit changes to be automatically included in the change representation:

- If a concept is added or removed as a child of an existing concept and the existing concept or one of its parents has a restriction on a property then a restriction change is made on the added or removed concept implicitly.
- If a property is added or removed as a child of another property then the domain and range of the added or removed property is changed implicitly.

4.5.3 Implementation

We have implemented the change representation approach using the Jena[4] Semantic Web framework. The OWL representation of the proposed meta ontology of figure 4.4 simply consists of classes for concepts, properties and instances as well as properties that indicate the relations. We use the properties *subClassOf*, *subPropertyOf* and *instanceOf* to define the class and property hierarchy and the instance of relation. In order to distinguish the relations for the old and new ontology version all those properties exist with postfixes *New* and *Old* in order to describe, that the relations holds in the old or the new ontology version. These properties describe only the direct relations between classes. In order to express the transitive relations, properties with the additional postfix Inf are used. The computation of those inferred properties can be realized by simple rules over the meta ontology instances. An example rule for the

[4]http://incubator.apache.org/jena/

inferred subclasses is the following: *subClassOfNew(?b, ?a) , subClassOfNew(?c, ?b) -> subClassOfNewInf(?c, ?a).* The meta ontology also contains properties for property restrictions, domain and range of properties and property assertions of individuals. Each hierarchical relation has an inverse relation (eg. *subClassOf/SuperClassOf*)

The implementation of the change representation operates in multiple phases:

- Classify the old version O_n and the new version O_{n+1} using the pellet reasoner.
- Transform O_n and O_{n+1} to instances of the meta ontology MO_n and MO_{n+1}.
- Generate changes by querying MO_n and MO_{n+1} together with explicitly provided knowledge about renames, merges and split operations.
- Store the found changes in form of instances of the change ontology. The changes relate instances of MO_n and MO_{n+1}.

Meta Ontology Transformation: The transformation step is realized by iterating over all classes and properties of the classified input ontologies. We use the Jena API and the pellet reasoner for this purpose. For each class the relations to its direct subclass are added to the meta ontology using the property assertion *subConceptOfOld* for the old ontology version and *subConceptOfNew* for the new ontology version. The sub-property hierarchy is transformed analogously using the properties *subPropertyOfOld* and *subpropertyOfNew*. During this iteration also the property assertions for the meta ontology instances, for the domain and range of properties as well as for property restrictions are added in form of property assertions. In addition the instances are generated and linked to their direct concepts with the property assertion $instanceOfOld$ and $instanceOfNew$ respectively.

Change-Generation: We treat the rename, merge and split operations as known a priori. Therefore, our implementation cannot construct them. Instead the rename operations are used to guarantee that elements with the same URI in both ontologies refer to the same elements. This is realized with an additional preprocessing step, that renames all URIs of the old version to the URIs in the new version. In practical scenarios the rename, split or merge operations can either be provided by the used ontology management system or by using change-detection algorithms such as [71] or [27]. The generation of all other changes is realized by simple SPARQL queries over the meta ontology instances. We provide an example for added subconcepts in listing 4.1. It basically queries for all subclasses that are subclasses of some concept in the new version that were not subclasses of that concept in the old version: $subClassOfNew(?sub, ?super) \wedge \neg\ subClassOfOld(?sub, ?super) \Rightarrow addChildConcept(?sub, ?super)$. Only the limited support for negation makes the SPARQL query a bit longer using the filter predicate.
The composite operations merge and split are directly added as instance of the change ontology. In addition, all atomic change-operations that are used to implement the merge or split (such as delete or add) are queried using simple SPARQL queries. The resulting atomic changes are than enhanced with a property assertion of the property $partOfComplexChange$

to the specific instance of the composite change operation.

```
1 SELECT ?sub ?super
2     WHERE {
3         ?sub subClassOfNew ?super.
4         OPTIONAL { ?y subClassOfOld ?super . FILTER (?sub = ?y).}
5         FILTER ( !BOUND(?y) )
6         };
```

Listing 4.1: Example SPARQL query for the detection of added subconcepts

Generated Output: The generated output of our implementation is an ontology that contains both, the instances of the meta ontology of the old and new ontology version and the detected changes. This allows to use standard technologies such as SWRL or SPARQL to query both the changes as well as the consequences with regard to the old and the new ontology version. This output is later used for the structural maintenance of annotations as well as for the detection of semantic changes. An example of the generated output can be found in figure 8.6 of the case-study in chapter 8.

Our change-detection implementation turned out to provide a very good performance (less than a minute of comparison time) even, when big ontologies with ten-thousands of concepts were compared.

4.6 Conclusion

In this chapter we have first discussed the requirements for the change representation for the structural, logical and semantic maintenance of annotations. We have then provided a survey on approaches that deal with ontology changes. The approaches can be divided into two groups: Approaches that operate on frame-based ontology formalisms and approaches that operate on DL-based modeling formalisms. The frame-based approaches typically represent changes in form of operations over the ontology-graph. This kind of representation is directly useful for the structural maintenance of annotations. In case of OWL ontologies the typical form of change representation is an axiom-LOG that defines what axioms were added or removed. This kind of change representation can be used to create the new ontology version from the old one but it does typically not explicitly express the semantics of the changes. Because our approach is based on OWL but we still need declarative change operations and reasoning support over the ontologies and changes we have decided to combine the best of both worlds with a hybrid approach. In this approach the input OWL ontologies are classified and then transformed to instances of a meta ontology that focusses on the structure of named entities of the ontology. On this representation the changes are identified and stored as instances of a change ontology. This form of representation allows to query the changes

in combination with both ontology versions. This high-level change representation approach can be accomplished with an additional standard axiom-based log-file to enable ontology versioning and to assist the justification of logical invalidations of annotations. We have finally presented the implementation of the proposed change representation using standard Semantic Web frameworks.

Chapter 5

Structural Maintenance of Annotations

The evolution of the reference ontology has consequences for the validity of the annotations and the interpretation of the schema elements. When the reference ontology evolves the semantic annotations need to be maintained. We distinguish three different kinds of invalidations: Structural, logical and semantic invalidation (detection of semantic changes). In this chapter we concentrate on the structural maintenance of annotations. We first present change operations on annotation paths that can be used to maintain invalid paths in section 5.1. In section 5.2 we define structural invalidations and propose an algorithm to detect such structural errors. Section 5.3 presents atomic and composite evolution strategies for structurally invalid paths and proposes an algorithm for the automatic computation of possible repair actions. The algorithm and the evolutions strategies are based on the change representation from chapter 4.

5.1 Annotation Change Operations

The goal of annotation change operations is to modify annotation path expressions. On the one hand such operations can directly be integrated into an annotation maintenances system in order to process changes. On the other hand a LOG of changes can be used as a basis for a versioning system for annotations.

5.1.1 Problem Definition

Given a set of annotation path P that are used for a specific XML-Schema S we need a set of operations that allows to change any annotation p in P to a transformation of p that we call p'. p' can be a variant of p or a totally new annotation. Every annotation path in P consists of a

sequence of steps that refer to concepts or properties of the reference ontology O. Each step s in p has a unique position and a type.

5.1.2 Local Changes

We divide the change operations into local and global change operations. A local change operation changes one specific step of an annotation path p. Each local change operation has a reference to the changed annotation path and a position *pos*. The position defines the step that is subject to change. It is provided by an index starting at 0.

- $AddStep(path, pos, step)$ - Adds a new step at position *pos*.
  path[pos] = step; All steps with an index equivalent or greater than *pos* are shifted to the right.
- $RemoveStep(path, pos, step)$ - Removes a step at position *pos*.
  All steps with an index equivalent or greater than *pos* are shifted to the left.
- $UpdateStepURI(path, pos, URI)$ - Changes the URI of the step at position *pos* to URI.
  path[pos].ref = URI;
- $UpdateRestriction(path, pos, restriction)$ - Changes the restriction of a concept-step at position *pos* to *restriction*.
  path.pos.restriction = restriction

Completeness of the Change-Operations:
By providing add, delete and update operations for steps every annotation path can be transformed to any other annotation path. Therefore, the change-operations are complete.

5.1.3 Global Changes

A global change operation is not defined for a specific path but for a set of path. The set of path P is typically the set of annotation path of one schema. Every global change operation can also be implemented with a set of local change operations. Due to the fact that typically many paths are effected by a change in the reference ontology it is valuable to have global operations in order to achieve a compact change-log.

- $UpdateGlobal(P, OldURI, NewURI)$ Changes all usages of $OldURI$ to $NewURI$
  $\forall s \in \{\forall p \in P\} \mid s.ref = OldURI \Rightarrow s.ref = NewURI$

5.1.4 Change Transactions

The addition and removal of steps can lead to structurally invalid annotation paths. Therefore, multiple changes need to be made in order to achieve a structural valid annotation path. We

propose to solve this issue by using transactions. The structural validity of a path only needs to be guaranteed, when a transaction has finished. In order to provide transaction support we define the following statements:

- $beginTransaction(path)$ - Begins a transaction for the path $path$
- $endTransaction(path)$ - Ends a transaction for the path $path$. If the path is structurally valid, then all operations are applied. Otherwise all changes are skipped.

5.2 Structural Invalidation of Annotation Paths

In order to discuss the structural invalidation, we will first briefly recall the definition of annotation paths: The set of all annotations of a specific schema is denoted P. Each annotation path $p \in P$ consists of a sequence of steps S. Each step $s \in S$ has a reference to an ontology element $s.ref$, a type $s.type$ and an optional restriction. The type directly depends on the referenced element in the ontology O. The ontology consists of a set of concepts C, a set of object-properties OP and a set of datatype-properties DP and a set of individuals I. Each $e \in \{C \cup OP \cup DP \cup I\}$ has a URI that uniquely identifies the ontology element. The corresponding sets of URI's are denoted with $C.URI$, $OP.URI$ and $DP.URI$. The type of step is determined by the referenced element in the ontology:

$\forall s \in S : s.ref \in C.URI \Rightarrow s.type = ConceptStep$
$\forall s \in S : s.ref \in OP.URI \Rightarrow s.type = ObjectPropertyStep$
$\forall s \in S : s.ref \in DP.URI \Rightarrow s.type = DatatypePropertyStep$

A step can be addressed by its position in the path. The position is defined by $s.pos$. The first step has the position 0. An annotation path has a length $p.length$ that corresponds to the number of steps. In order to form a structurally valid annotation path a number of constraints over the sequence of steps exist.

Definition 5.2.1. *Structurally Valid Annotation Path*
An annotation pas p is structurally valid, if the following conditions hold for each step s of p:

- The first step $(s.pos = 0)$ must be a $ConceptStep$.
- The last step $(s.post = p.length - 1)$ must be a $ConceptStep$ or a $DatatypePropertyStep$.
- An $ObjectPropertyStep$ must only exist between two $ConceptSteps$.
- A $ConceptStep$ must be followed by an $ObjectPropertyStep$ or a $DatatypePropertyStep$ or nothing.
- A $DatatypePropertyStep$ can only exist as the last step.

- Only a *ConceptStep* may have a restriction.

An annotation path that does not match these conditions is a structurally invalid annotation path. Every structural invalidation of an annotation path is caused by at least one invalid step. There are two types of invalid steps:

Definition 5.2.2. *Missing-Reference Invalidation*
A step s of an annotation path p is invalid due to a Missing-Reference Invalidation, iff
$s.ref \notin O.URI$

Definition 5.2.3. *Wrong-Type Invalidation*
A step s of an annotation path p is invalid due to a Wrong-Type Invalidation, iff:
$s.ref \in O.URI \wedge$
$(s.type = ConceptStep \wedge s.ref \notin C.URI) \vee$
$(s.type = ObjectPropertyStep \wedge s.ref \notin OP.URI) \vee$
$(s.type = DatatyPropertyStep \wedge s.ref \notin DP.URI)$
The required type is defined by the structural constraints from definition 5.2.1

These constraints can directly be transformed to an algorithm that checks the structural validity of an annotation path. The proposed algorithm in listing 5.1 returns an error-report, that contains information about the positions and types of errors. If no error was found an empty error-report is returned[1].

```
1  checkStructValid(annotationPath p, Ontology O) {
2      errorReport new ErrorReport();
3      for (i=0; i<=p.length; i++) {
4          if (!O.isUriInOntology(p[i].ref)
5              errorReport(i,p[i].ref,'Missing-Reference')
6          else if (i%2==1 and i!=p.length-1 and !O.isObjectProperty(p[i].ref))
7              errorReport(i,p[i].ref,'Wrong-Type','Object-property required')
8          else if (i%2==0 and !O.isConcept(p[i].ref))
9             errorReport(i,p[i].ref,'Wrong-Type','Concept required')
10        }
11     if (p.length-1%2==1 and !O.isDatatypeProperty(p[p.length -1].ref))
12        errorReport(i,p[i].ref,'Wrong-Type','DataType-Property required')
13 return errorReport();
14 }
```

Listing 5.1: Structural validation of an annotation path

The methods $isObjectProperty(uri)$, $isDatatypeProperty(uri)$ and $isConcept(uri)$ of an ontology return *true*, if *uri* is an object-property/datatype-property/concept in the reference

[1] The depicted algorithm does not check for errors that are caused by restriction paths. This can be realized analogously by iterating over the steps of the restrictions. In this case also references to instances must be checked.

ontology. The method $isUriInOntology(uri)$ returns *true*, if *uri* is defined in the reference ontology.

5.3 Evolution Strategies for Structurally Invalid Paths

In section 5.2, we have defined when an annotation path gets structurally invalid. In order to decide how a structurally invalid path can be repaired the reasons for the invalidation needs to be investigated. Sources of knowledge for possible evolution strategies are the old and the new ontology version and the change representation of chapter 4.
The evolution strategies are inspired by the evolution strategies for ontologies as proposed in [37] and are related to the evolution strategies for RDF data that is structured by an evolving RDF-Schema in [62].

Problem Definition: Given an annotation path p_i, two succeeding versions of the reference ontology O_n and O_{n+1} and the representation of changes between O_n and O_{n+1}, where the path p_i is structurally valid with regard to O_n and p_i is structurally invalid with regard to O_{n+1}. Find a variant of p that is valid in O_{n+1}. The variant should preserve as much of the semantics as possible.

5.3.1 Atomic Evolution Strategies for Missing-Reference-Invalidations

Every structural invalid annotation path p contains at least one invalid step s. When all invalid steps are repaired, then p is structurally valid. A $MissingRefInvalidation$ of a step s in an annotation path p is a local invalidation, where it is sufficient to replace the reference of the step $s.ref$ with an element that is existent in O_{n+1}. Which ontology element can be used as a replacement depends on the type of change that occurred. Therefore, we will discuss the possible evolution strategies for each type of change in the ontology. We describe the strategies in form of rules. The predicate $MissingReferenceInvalidation(?step, ?path)$ defines that the step *?step* is invalid in the path *?path* due to a missing-reference invalidation. The predicate *changeAnnotation(?mtid,?tid)* defines that the change with the id ?*tid* is a member of a composite change with the id ?*mtid*.

Rename of Ontology Elements

If a step of a path is invalid because the URI of the referenced element has changed, then the new URI can be used for the step:

$MissingReferenceInvalidation(?step, ?path) \wedge rename(?tid, ?step.uri, ?to)$
$\Rightarrow UpdateStepURI(?path, ?step.pos, ?to)$

This evolution strategy is a fully semantics preserving operation, where no user-intervention is required.

Merge of Ontology Elements

If a step is invalid because the URI of the referenced element was deleted and the delete is associated with a merge operation, then the target of the merge can be used for the annotation. A merge is a composite change-operation that is linked to atomic change operations by change-annotation predicates.

$MissingReferenceInvalidation(?step, ?path) \wedge delete(?tid, ?step.uri)$
$\wedge\, merge(?mtid, ?from, ?to) \wedge changeAnnotation(?mtid, ?tid)$
$\Rightarrow UpdateStepURI(?path, ?step.pos, ?to)$

This evolution strategy is a fully semantics preserving operation in the sense that only semantics are lost that cannot be expressed in O_{n+1}. Thus, no user-intervention is required.

Split of Ontology Elements

If a step is invalid because the URI of the referenced element was deleted and the delete is associated with a split operation, then each target of the split is a possible replacement candidate. A split is a composite change-operation that is linked to atomic change operations by change-annotation predicates.

$MissingReferenceInvalidation(?step, ?path) \wedge delete(?tid, ?step.uri)$
$\wedge\, split(?mtid, ?from, ?to) \wedge changeAnnotation(?mtid, ?tid)$
$\Rightarrow UpdateStepURI(?path, ?step.pos, ?to)$

In contrast to the previous strategies the given rule may return multiple different possible change-operations for the repair. Since the more general concept that was used previously does not exist after the split operation the user needs to select a suitable operation that best fits the semantics of the schema element. If this is not possible the user may decide to use a parent element of the deleted one.

General Deletes

If a step is invalid because the URI of the referenced element was deleted and the delete is not associated with a split or merge operation, then a still existing parent element in O_{n+1} can be used.

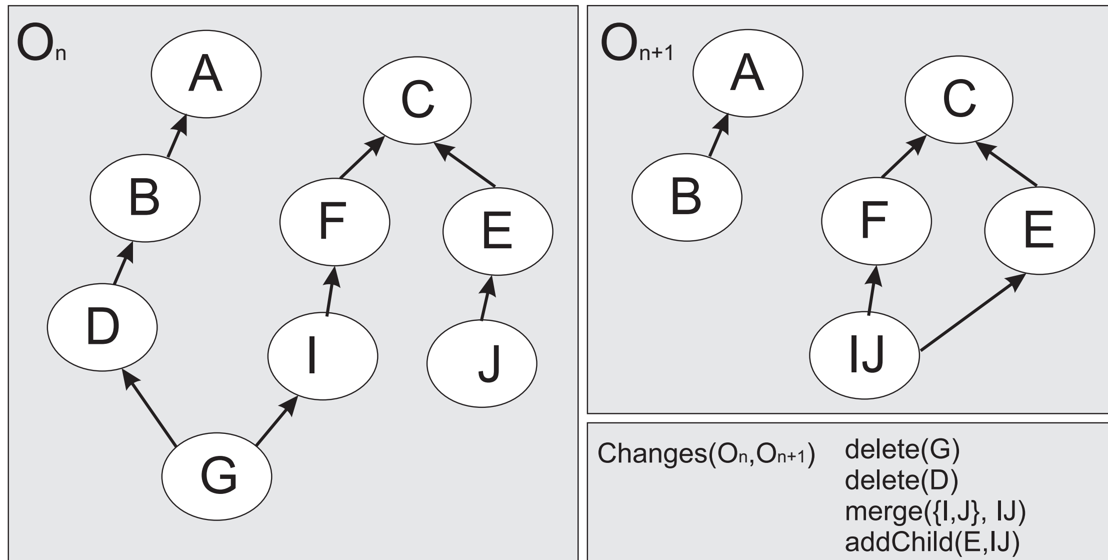

Figure 5.1: Example ontology for evolution strategies for structural invalidations

$MissingRefInvalidation(?step, ?path) \wedge delete(?tid, ?step.uri)$
$\wedge !changeAnnotation(?mtid, ?tid) \wedge hasDirectParent(?step.uri, ?parent)$
$\Rightarrow UpdateStepURI(?path, ?step.pos, ?parent)$

This strategy exploits the fact that each instance of a subconcept is also an instance of all its superconcepts and that any property assertion for a sub-property is also a property assertion for all its super-properties. Thus, it is semantically correct to use a parent element, if the child element does not exist in O_{n+1}. This strategy returns a more general annotation and it should be reviewed by the user. The predicate $hasDirectParent(?step.uri, ?parent)$ returns all direct parents of *step.uri* that are still existent in O_{n+1}. This includes that possibly no replacement can be found or that multiple elements may be returned.

5.3.2 Composite Evolution Strategies for Missing-Reference-Invalidations

We have defined atomic evolution strategies for structural invalidations of steps in subsection 5.3.1. They could either provide direct replacements or they provided more general ontology elements for the invalid step. The selection of a more general replacement can be *direct*, if the direct parent still exists or it can be *indirect*, if the direct parent was itself subject of a change. As a result the atomic strategies need to be combined to provide a suitable replacement for an invalid step. We will first provide an example for such a scenario.

A fragment of the class hierarchy of an example ontology with two versions is shown in figure 5.1. We assume the annotation path */X/has/G/has/Z*. This annotation is structurally valid in O_n, but structurally invalid in O_{n+1}. The cause for the invalidation is the missing concept G. Since G was deleted and this delete is not associated with a composite operation we need to find an existing parent of G in O_{n+1}. Unfortunately, the former parents of G, D and I are also not existent in O_{n+1}. The parent B of D exists and can be used as a replacement. In addition I was merged with J and is now called IJ in O_{n+1}. We get the following set of possible replacements for G:

1. Replace G with IJ. (1 Abstraction)
2. Replace G with B. (2 Abstractions)
3. Replace G with F. (2 Abstractions)
4. Replace G with A. (3 Abstractions)
5. Replace G with C. (3 Abstractions)

Since we assume that each abstraction reduces the specificity of the annotation we should use option 1, which provides a structurally valid solution with a minimum number of abstractions.

An Algorithm for the Repair of Missing Reference Invalidations

The previous example has shown that the repair of an invalid step may require a combination of atomic evolution strategies. For example, when an ontology element was deleted and the direct parent of it was merged, we need to use the target of the merge as a replacement for the deleted concept. In addition multiple solutions can exist. The quality of a solution can be ranked by the number of required abstraction steps.

Problem Definition: Given an existing ontology element $i \in O_n$ that is referenced from a structurally invalid step in O_{n+1} we need to find the set of replacement candidates J of i. Each replacement $j \in J$ has a value *j.val* that defines the replacement target in O_{n+1} and an indicator of the number of abstractions *j.abstr*. The best replacement candidates are those with a minimal value of *j.abstr*.

We model the search space in form of a tree, which we call the replacement-tree, where the missing element i forms the root. The children of the root elements are the parent elements of $i \in O_n$. In order to find all replacement candidates we can simply perform a breath first traversal over the replacement-tree starting at the root i and check each node for possible replacements according to the atomic evolution strategies. Whenever a solution is found we can add it to the result. The parameter *j.abstr* is equivalent to the level of the tree, where the solution was found. The corresponding algorithm for invalid concept steps is shown in listing 5.2.
The algorithm for property-steps can be implemented analogously. The algorithm gets the

invalid step, the old and the new ontology version and the change representation as input. The parameter *k* specifies the maximum number of results. By using a breath first traversal it is guaranteed that the best *k* replacements are returned.
The key method *getReplacementConcept()* for this algorithm is shown in listing 5.3. The method basically checks the applicability of the proposed atomic evolution strategies. When such a strategy is applicable the possible replacements are returned.

```
1  getReplacements(step s, Ontology o,Ontology on, Changes C, int k)
2  int f = 0;
3  rp = CreateReplacementTree(step s, o, 1);
4      breath-first traversal over all nodes n of rp {
5          candidate = getReplacementConcept(s,n,o,on,C);
6          if (candidate != OWL:Thing) {
7              result.add(candidate,n.level);
8              f++;
9              }
10         if (f > k) break;
11         }
12 return result;
```

Listing 5.2: Algorithm for the generation of candidate replacements for invalid concept-steps

The method uses the helper functions *hasConcept*, *hasRenameC*, *hasMergeC* and *hasSplitC*. The function *hasConcept* returns *true* if the new version contains a concept with the given URI. The functions *hasRenameC*, *hasMergeC* and *hasSplitC* return *true*, if the provided URI was subject to a rename, a merge or a split. If this was the case the corresponding *getRenameC*, *getMergeC* od *getSplitC* are used to get the corresponding element of the new ontology version. In case of *getSplitC* a set of solutions is returned. When no replacement is possible null is returned.

```
1  getReplacementConcept(step s, node n,Ontology o,Ontology on, Changes C) {
2      if (s.type==concept) {
3          if (on.hasConcept(s.ref))
4              return s.ref
5          else if (C.hasRenameC(s.ref))
6              return C.getRename(s.ref)
7          else if (C.hasMergeC(s.ref))
8              return C.getMergeTarget(s.ref)
9          else if (C.hasSplitC(s.ref))
10             return C.getSplitTargets(s.ref)
11         else return null
12         }
13     return 'wrong_type';
14 }}
```

Listing 5.3: Definition of getReplacementConcept()

5

Repairing an Invalid Path

We have proposed a method to find and rank possible replacements for steps that got invalid due to missing reference invalidations. Given a path, with only one invalidation, where the invalidation has the type missing-reference we can use the proposed algorithm to compute a number of replacement candidates. We can then generate a new annotation path for each replacement candidate. Each such path is structurally valid. In order to help the user to select an appropriate replacement path we can also validate it logically and only present logically valid solutions. If the user is asked can also depend on the quality of the solutions. For example a user-review is not required, when no abstraction step was necessary and the corresponding atomic evolution strategy returned only one solution.

We will now discuss the problem, when multiple steps of a path are invalid due to a missing-reference invalidation.

Given a path p with a set of invalid steps I. Each invalid step $i \in I$ is invalid due to a missing-reference invalidation. In addition we have a set of ranked possible replacements for each invalid step $i.REP$. Each element of $i.REP$ is a tuple *(replacement,abstr)*. We want to find a set of the best k candidate path P such that each p in P is structurally valid and the number of abstractions for each replaced step is minimized.

The set of all candidate solution path is defined by $CAND = i_1.REP \times i_2.REP ... \times i_n.REP$. Each candidate $\in CAND$ can be ranked according to the sum of all *abstr* values. Thus, we can generate the first k solutions that use replacements with minimal abstraction levels. The result can additionally be checked for logical invalidations and can then be presented to the user. If no abstractions were required for a valid solution and only atomic strategies that require no user intervention were used to generate the best result, then the user-review can be omitted.

5.3.3 Evolution Strategies for Wrong-Type Invalidations

The previous algorithm could be used to find a suitable replacement for paths with missing reference invalidations. The general idea is that the type of the URI was not changed. In case of wrong-type invalidations this is different. For example something that was modeled as datatype-property before is now modeled as an object-property. In case of annotation maintenance we always assume that an annotation path was valid in the previous ontology version. This means the structural constraints were met in the old ontology version. Therefore, a wrong-type invalidation is only possible in the following cases[2]:

1. Concept to object-property: $s.uri \in O_n.C.URI \wedge s.uri \in O_{n+1}.OP.URI$
2. Concept to datatype-property: $s.uri \in O_n.C.URI \wedge s.uri \in O_{n+1}.DP.URI$

[2]We assume that the URI was not changed, but the type of element in the ontology has changed. One might argue that when such a modeling change occurs, then also the URIs will change. If we suppose that the URI changes, then additional complex changes that represent the different cases are required in order to express such modeling changes.

3. Object-property to concept: $s.uri \in O_n.OP.URI \wedge s.uri \in O_{n+1}.C.URI$
4. Object-property to datatype-property: $s.uri \in O_n.OP.URI \wedge s.uri \in O_{n+1}.DP.URI$
5. Datatype-property to object-property. $s.uri \in O_n.DP.URI \wedge s.uri \in O_{n+1}.OP.URI$
6. Datatype-property to concept. $s.uri \in O_n.DP.URI \wedge s.uri \in O_{n+1}.C.URI$
7. Concept to instance: $s.uri \in O_n.C.URI \wedge s.uri \in O_{n+1}.I.URI$
8. Instance to concept: $s.uri \in O_n.I.URI \wedge s.uri \in O_{n+1}.C.URI$
9. Object-property to instance: $s.uri \in O_n.OP.URI \wedge s.uri \in O_{n+1}.I.URI$
10. Datatype-property to instance: $s.uri \in O_n.DP.URI \wedge s.uri \in O_{n+1}.I.URI$
11. Instance to object-property: $s.uri \in O_n.I.URI \wedge s.uri \in O_{n+1}.OP.URI$
12. Instance to datatype-property: $s.uri \in O_n.I.URI \wedge s.uri \in O_{n+1}.DP.URI$

This list is complete because each relevant ontology element of an annotation can be changed to any other type of ontology element. In contrast to the missing-reference invalidation the semantics of wrong-type invalidations are less strict and their resolution typically requires human intervention. Therefore, we will concentrate on a limited set of transitions between types that have strict semantics: (5) datatype-property to object-property, (4) object-property to datatype-property, (7) concept to instance and (8) instance to concept.

Datatype-Property to Object-Property

According to the old ontology version the last step referred to a datatype-property. In the new ontology version an object-property is referenced. In order to repair this structural problem we can use the required object-property as it is. But we need to add a new concept-step to comply with the annotation method. According to the new ontology version any concept that is in the range of the object-property and that is not disjoint from the range of a restriction on that property on the the previous concept-step can be used. Obviously the user needs to select one concept out of a list of possible concepts. In order to get a structurally valid annotation concept also the usage of *Thing* is possible.

Object-Property to Datatype-Property

In this case a combination of an object-property and a concept at the end of an annotation path gets invalid because the object-property is a datatype-property in the new ontology version. Thus, only the last step, which must be a concept-step needs to be removed. The new last step refers to the datatype-property. This solution can only be applied, when the changed-property is at the last position of the annotation path.

Concept to Instance

The reference to an instance in an annotation path is only allowed in the restriction of concept-steps. We therefore, need to find a replacement for the missing concept and add a restriction for the specific individual. Given a uri c that refers to a concept C in O_n and to an instance I in O_{n+1} we can replace the *uri* with a concept $C2$ of O_{n+1}, where I is an instance of $C2$. In order to preserve the semantics a restriction on that concept in the annotation path for the individual I must be created.

Instance to Concept

In this case there is a restriction on a concept step to an instance that is now a concept. This can be repaired by replacing the concept of the concept-step with the *uri* of the former individual and the removal of the individual in the restriction.

5.4 Annotation Maintenance using Mapping Composition

In the last sections we have shown how a structurally invalid annotation path can be repaired, when both ontology versions and a representation of changes are given. This task requires partly human intervention, which should be minimized as much as possible. Since semantic annotations can be used to match schemas of different partners (see chapter 3) those matches can be an additional source of knowledge. This leads to the following problem:

Given an invalid annotation path p of some element $e1$ of some schema $S1$ and a set of schemas S. Each schema s in S is annotated with a set of schema annotations $s.A$. Each schema annotation $sa \in s.A$ consists of a set of annotation path $sa.P$ and has a corresponding version of the reference ontology $sa.O$. We need to find an equivalent class match between $p \in S1$ and some other annotation path $p2$ of some other schema $S2$, where $p2 \in sa1.P$ and $\{sa1, sa2\} \subseteq S2.A$ and $sa1.O = O_n$ and $sa2.O = O_{n+1}$. When such a match between p and $p2$ exists, then there is a schema element $e2$ that is annotated with $p2$ and the annotation $p3 \in sa2.P$ of $e2$ is a replacement candidate p' for p. The scenario is shown graphically in figure 5.2.

The match between p and $p2$ is based on the standard matching methods from chapter 3. This annotation upgrade method is based on the ideas of mapping composition [2, 34]. We suppose that it allows a dramatic reduction of manual annotation maintenance effort since theoretically each annotation with equivalent semantics needs to be maintained only once. In addition such a cooperative annotation maintenance approach has the advantage that all partners will use the same replacement for equivalent invalid annotations, which reduces the differences between annotations for semantically equivalent entities.

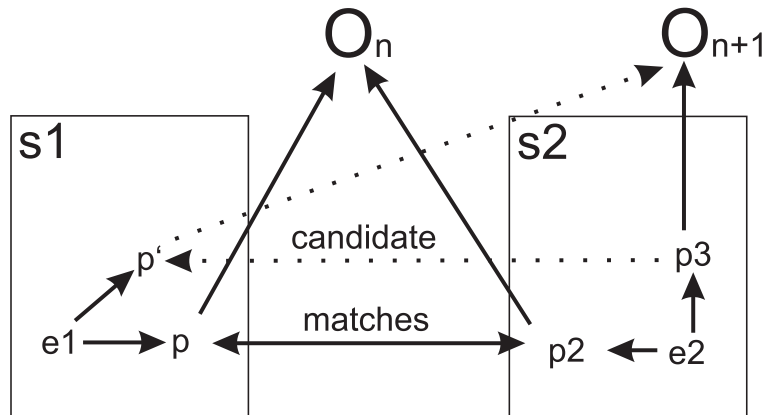

Figure 5.2: Annotation maintenance using mapping-composition

5.5 Conclusion

In this chapter we have first introduced a complete set of change operations for the modification of annotation paths. Those change operations can be used to repair annotation paths that got invalid due to ontology changes. Structural invalidations occur, when named entities of the ontology that are referenced from an annotation path step do not exist in the new ontology version or when the type of the referenced ontology element has changed. We have proposed an algorithm for the detection of such invalidations. An invalid path consists of at least one invalid step. An invalid step that references a missing element can be repaired by replacing the referenced element with an existing element of the new ontology version. Such a replacement can be computed by evaluating the changes between the old an the new ontology version and may require to replace a more specific element of the old ontology version with a less specific element of the new ontology version. This property is used to rank possible replacements. The overall goal is to find replacements with minimal semantic changes. When the possible repairs for each invalid step of a path are computed those solutions can be used to propose replacement candidates for the complete annotation path. We propose that if no fully information preserving replacement could be found, the user should review a set of k best annotation path repair candidates and select an appropriate one. The set of best repairs are those with a minimal overall number of abstractions for all replaced steps. The set of possible replacement path can also be filtered by first checking its logical validity, which is discussed in the next chapter. We have finally introduced an additional annotation maintenance scenario in a collaborative environment that limits the human intervention by using mapping composition.

Chapter 6

Logical Invalidation of Annotations

6

[1]We have discussed the structural invalidation and maintenance of annotation in the last chapter. Basically an annotation path is structurally invalid, if it does not fulfill the structural requirements of the annotation method. The result is that the annotation path cannot be transformed to a structurally valid OWL concept. As a consequence every structurally valid annotation path can be transformed to an OWL concept but it is not guaranteed that this is a satisfiable concept with regard to the reference ontology.

There are two scenarios, where annotations need to be validated against the reference ontology: Annotation creation and annotation maintenance. In both cases the task requires additional domain-knowledge: In case of the creation the annotator needs to define suitable and valid annotations for the schema elements. In case of annotation maintenance the user needs to upgrade invalid annotations to annotations that are valid with the new ontology version. Because ontologies are typically refined during evolution the old ontology version can be seen as an under-specification of the current real-world domain. As a consequence the maintenance of the annotations is a non-trivial task, where additional domain knowledge is required. Thus, the task of repairing logically invalid annotation needs to be performed by human experts. Nevertheless, those experts need support to find suitable solutions. In this chapter we will define invalidation-types and propose black-box methods and algorithms that can precisely track those invalidations in annotation paths. This information can be used by a user to repair the path expressions. We concentrate on black-box approaches because one major usage scenario of semantic annotations are heterogeneous information systems. In such a scenario it is not realistic to enforce the usage of a specific reasoner.

[1]Parts of this chapter have been published in [59]

6.1 Annotation Method

In order to discuss why an annotation can be logically invalid we will first briefly recall the annotation method from chapter 2. An annotation of some XML-Schema element or type is a path expression that consists of a sequence of steps. Each step corresponds to either a property or a concept of the reference ontology. Structurally valid annotation path expressions can automatically be transformed to OWL2 [75] concepts. Such concepts are used to represent the semantics of the annotated element. We will describe this with an example:
The annotation path $p =$ *Order/billTo/Buyer[Mr_Smith]/hasCountry/Country* could be used to annotate a *country* element for some XML-Schema for *order* documents. It describes a subconcept of a *country* that has an inverse relation *hasCountry* to some *Buyer* that has an inverse *billTo* relation to some *Order*. The buyer has a restriction to state that the Buyer is a specific buyer with the name *Mr. Smith*. The corresponding class definition *p.c* is shown in listing 6.1.

```
1 Class: Order/billTo/Buyer[Mr_Smith]/hasCountry/Country
2 EquivalentClasses(
3      ConceptAnnotation and Country and inv
4        (hasCountry) some
5           (Buyer and {Mr_Smith} and inv (billTo) some (Order)
6        ))
```

Listing 6.1: Representation of an annotation path in OWL

In order to transform a structurally valid annotation path expression to an OWL concept the following mappings apply:

- *Concept-Steps* are directly mapped to concepts.
- Restrictions on *Concept-Steps* are mapped to enumerated classes or restrictions over the corresponding concept.
- *Object-property-Steps* are mapped to inverse *some values from* restrictions between concepts on that specific property.
- *Datatype-Property-Steps* are mapped to *some values from restrictions* of the last concept step on that specific datatype-property.

The annotation method allows different types of annotations, that we will now define in order to describe the possible invalidations for each type in the following sections.

- *Simple Concept Annotations* consists of only one concept.
- *Simple Datatype Annotations* consists of only one concept and one datatype-property.
- *3-Step Concept Annotations* consists of a concept, an object-property and another concept.
- *General Annotations* are annotation paths that consists of more than 3 steps.

6.2 Logical Invalidation of Annotation Paths

An annotation path is logically invalid, if the corresponding annotation concept is not satisfiable in the reference ontology. Thus, the detection of logically invalid annotations is a classical reasoning task. The root concept of OWL is *Thing*. Any subconcept of this concept is satisfiable in the ontology. A satisfiable concept can contain individuals. Concepts that are not subclasses of *Thing* are not satisfiable and are subclasses of the concept *Nothing*, which is the complement of *Thing*.

Definition 6.2.1. ***Logical Invalidation of an Annotation:*** A structurally valid annotation path p is logically invalid if the corresponding annotation concept $p.c$ is unsatisfiable in the reference Ontology O. $O \bigcup p.c \rightarrow p.c \not\sqsubseteq \text{Thing}$

6

Why can't we use standard tools for validation and repair such as [77] or [89]? First, we want to determine which steps of the annotation path are responsible for the invalidation rather than determining a set of axioms of the (extended) ontology causing the invalidation. Second, repairs can only change annotation paths, and not axioms of the ontology. For debugging annotations we have the following requirements:

- The ontology is assumed to be consistent and therefore, free of contradictions before the annotation concept is added.
- The structure of the annotation concepts is strictly defined by the annotation method.
- Repairs can change annotation path expression but not the ontology.
- In case of annotation maintenance we require that the annotation concept was valid in the previous ontology version.

Therefore, we need to find the error in the steps of the annotations rather than in their OWL representation. This limits the usefulness of standard OWL debugging methods (see section 6.6). If an ontology evolves, annotation maintenance means to identify those annotation paths which became logically invalid due to the changes in the ontology and to identify those steps in the annotation path which cause the invalidation. An expert then can repair the invalid annotation paths efficiently using the information about the cause of the invalidation.

In OWL logical contradictions boil down to a limited set of contradictions [77]:

- **Atomic** - An individual belongs to a class and its complement.
- **Cardinality** - An individual has a max cardinality restriction but is related to more distinct individuals.
- **Datatype** - A literal value violates the (global or local) range restriction of a datatype-property.

These clashes also apply for unsatisfiable classes. Thus, for example a class is unsatisfiable if it is defined as an intersection with its complement or if it has contradicting cardinality- or datatype-restrictions. Of course such invalidations can be produced by non-local effects. In the next sections we discuss how the different annotation types can be logically invalid.

6.2.1 Invalidation of Simple Concept Annotations

A simple concept annotation consists of only one concept. Thus, a concept with the name $prefix + conceptUri$ is generated, where prefix is some unique identifier that is not used in the ontology O, with the equivalent class definition (*ConceptAnnotation and conceptUri*).

Theorem 6.2.1. A simple concept annotation that is structurally valid is also logically valid.

Proof. We require that all concepts of the reference ontology are satisfiable. Thus, there is only one case, where the union of $ConceptAnnotation$ and *conceptURI* can result in an unsatisfiable concept: The class with the URI *ConceptAnnotation* is disjoint from the concept with the URI $conceptURI$. This is impossible because the primitive concept $ConceptAnnotation$ does not exist in the reference ontology before the annotations are added. Thus, there cannot be an axiom in the ontology that contradicts with it. □

6.2.2 Invalidation of Simple Datatype Annotations:

A simple datatype annotation of the form $/c/datatypeProperty$ consists of a concept and a restriction over some datatype-property of the form: *(datatypeAnnotation and c and datatype-Property some rdf:Literal)*.

Theorem 6.2.2. There exists no invalid simple datatype annotation that does not violate one of the following conditions:

1. ***Invalid-domain:*** The intersection of the domain of the property with the concept is not a subclass of $OWL : Thing$.
   $c \sqcap domain(datatypeProperty) \not\sqsubseteq Thing$
2. ***Invalid-restriction:*** The intersection of the concept and the restriction over the datatype-property is not a subclass of $OWL : Thing$.
   $c\ and\ datatypeProperty\ some\ Literal \not\sqsubseteq Thing$

Proof. Obviously, case 2 of an invalidation is equivalent to the satisfiability-check of the whole annotation concept. There is only one additional case for an invalidation where the concept with the URI $datatypeAnnotation$ is disjoint from c, which is impossible in analogy to theorem 6.2.1. Thus, every logically invalid simple datatype annotation is captured. □

According to theorem 6.2.2 every simple datatype annotation that is invalid due to an *invalid-domain* is also invalid due to an *invalid-restriction*. Thus, in order to detect the cause of the error in more detail we need to investigate the reasons for the invalid restriction. This can be realized by additionally checking the first case. In addition the restriction clash is not yet atomic. In OWL there are the following scenarios for invalid restrictions over datatype-properties:

1. The datatype of the restriction does not comply with a datatype that is required by an existing restriction in O.
2. There is a cardinality clash between the existential restriction of the annotation path and an existing restriction in O.

Theorem 6.2.3. An invalidation of a simple datatype annotation due to a conflicting datatype restriction is impossible.

Proof. A contradicting datatype must be disjoint from the datatype in the existential restriction. This is impossible because every datatype is a subtype of *rdfs:literal*, which is used for the existential restriction in the annotation concept. No subtype can be disjoint from its supertype. □

Cardinality clashes are possible, when there is a restriction on the class $(c \sqcap datatypeProperty)$ of the form: $datatypeProperty\ max\ n\ type$, where $type$ is $rdfs : Literal$ or any subtype of it.

6.2.3 Invalidation of 3-Step Concept Annotations

A 3-step concept annotation is a triple of the form *concept/property/otherconcept*. It is represented as an OWL equivalent class expression *otherconcept and inv (property) some concept*. Such an expression can be invalid due to *domain-invalidation*, *range-invalidation* and *restriction-invalidation*.

Definition 6.2.2. *Domain-Invalidation:*
An annotation triple of the form $concept/Property/otherconcept$ is unsatisfiable due to a *domain-invalidation*, iff: $domain(Property) \sqcap concept \not\sqsubseteq Thing$

Definition 6.2.3. *Range-Invalidation*
An annotation triple of the form $concept/Property/otherconcept$ is unsatisfiable due to a *range-invalidation*, iff: $range(Property) \sqcap otherconcept \not\sqsubseteq Thing$

Definition 6.2.4. *Restriction-Invalidation:*
An annotation triple of the form $concept/Property/otherconcept$ is unsatisfiable due to a *restriction-invalidation*, iff: $otherconcept \sqcap inv\ (Property)\ some\ concept \not\sqsubseteq Thing$

Theorem 6.2.4. There exists no invalid *3-step concept annotation* that does not introduce a *domain-invalidation*, *range-invalidation* or *restriction-invalidation*.

Proof. A *restriction-invalidation* is defined as $otherconcept \sqcap inv\ (hasProperty)\ some\ concept \not\sqsubseteq Thing$. This is equivalent to the satisfiability requirement for the whole annotation path because the intersection of $otherconcept$ and $ConceptAnnotation$ cannot result in a clash (see proof of theorem 6.2.1). Thus, there exist no invalid 3-step annotations that are not captured by the enumerated invalidations. □

While the domain or range invalidations are already atomic there can be different causes for invalid restrictions: A restriction can be invalid because the range of the restriction is disjoint from another $allvaluesFrom$ restriction on *concept* or it can be invalid because there is a cardinality restriction on *concept* of the form $property\ max\ n\ otherconcept$. Therefore, the *invalid-restriction* problem can be divided into *invalid-value-restriction* and *invalid-cardinality-restriction*.

OWL2 allows the definition of object properties to be functional, inverse functional, transitive, symmetric, asymmetric, reflexive, and irreflexive. Since the existence of the object-property is defined by the existential quantification of the inverse of the property these characteristics can influence the satisfiability of the annotation. For example, given an annotation path $p = /A/hasB/B$, the path is invalid, if $hasB$ is defined as inverse functional and B has an inverse $hasB$ restriction in O to some other class that is disjoint from A.
We can summarize that a *3-step concept annotation* can be invalid because of the restrictions that are formulated over the corresponding annotation concept. Definition 6.2.4 is sufficient but the root cause can be found in property characteristics or cardinality or value clashes.

6.3 Invalidation of General Annotations

In the last section we defined all local invalidations that can occur in annotations that consists of 3 steps. A general annotation consists of a sequence of 3-step concept annotations called triples. The last step can be a 3-step concept annotation or a simple datatype annotation. We will first show that all local invalidation types also apply to general concept annotations and then discuss additional kinds of invalidations that are only possible in general annotations.

6.3.1 Invalidation of General Annotation due to Local Invalidations

Definition 6.3.1. *Local-Invalidations:* The invalidation types *domain-invalidation* (see def. 6.2.2), *range-invalidation* (see def. 6.2.3) and *restriction-invalidation* (see def. 6.2.4) are local invalidations, that are defined in the context of a triple.

Theorem 6.3.1. A locally invalid 3-step annotation cannot get valid, when it occurs as a triple in a general annotation path.

Proof. A general annotation path has the form: $/c_1/p_2/c_3/.../c_{n-2}/p_{n_1}/c_n/$. We now assume that there exits a triple $C_{inv} = c_x/p_y/c_z$, in the path that is invalid, when it is inspected separately (local invalidation), but the entire annotation concept ... $c_{-2}/p_{-1}/c_x/p_y/c_z/p_1/c_2$ ... is valid. This implies that either c_x or c_z were implicitly changed to classes that are not still causing local invalidations in C_{inv}. When the triple C_{inv} is added to the annotation concept this is realized by an expression of the form:
... c_2 *and* (inv) p_1 *some* $(c_z$ *and inv* (p_y) *some* $(c_x$ *and* p_{-1} *some* ...
Thus, z_x is implicitly replaced with an intersection of z_x and $(p_1$ *some* ...$)$ that we now call z_{x2}. c_z gets implicitly replaced with c_z and $($*range* (p_1) that we now call c_{z2}. In order to achieve a satisfiable triple C_{inv} in p, c_{x2} must not be a subclass of c_x or c_{z2} must not be a subclass of c_z. This is a contradiction because they are logically subclasses of c_x and c_z. □

Theorem 6.3.2. A general concept annotation that contains an invalid triple is itself logically invalid.

Proof. A general concept annotation path consists of triples: $t_1/t_2/t_3/.../t_n$. We will now show via induction that as soon as one of its triples is unsatisfiable, the whole annotation concept is unsatisfiable. Beginning with an annotation p_1 that only consists of t_n. If t_n is itself unsatisfiable, then the whole path cannot be satisfiable because it is represented as a subclass of t_n in $p_1.c$. We now assume that p_1 is satisfiable and we add t_{n-1}, which is supposed to be unsatisfiable. The addition renders the whole annotation path unsatisfiable because the connection between p_n and t_{n-1} is represented in form of an existential restriction. This step can be repeated by adding an unsatisfiable triple to a longer and longer valid path, until t_1 is reached. Therefore, if any triple of a general concept annotation is locally invalid the whole annotation concept must be logically invalid. □

As a conclusion all previously discussed local invalidations also apply to general annotations. Additionally there are invalidations that only occur in general annotations: direct-triple-disjointness and arbitrary non local invalidations.

Figure 6.1: Example of direct-triple-disjointness

6.3.2 Direct-Triple-Disjointness

One kind of invalidation that does not exist for 3-step annotations can be caused by the concatenation of two annotation triples. This means the concept that is implicitly created by the first triple is disjoint from the concept which is required by the second triple. An example for such a scenario is shown in Figure 6.1. The corresponding reference ontology is shown in listing 6.2.

```
1 Order isA Document
2 Invoice isA Document
3 Disjoint(Order, Invoice)
4 Domain(SendsOrder) = Customer
5 Range(SendsOrder) = Order
6 Domain(hasInvoiceNumber) = Invoice
7 Range(hasInvoiceNumber) = InvoiceNumber
```

Listing 6.2: Example ontology for direct-triple-disjointness invalidations

In *Customer/sendsOrder/Document/hasInvoiceNumber/InvoiceNumber* each triple is valid individually, but the combination of the triples leads to an unsatisfiable concept. The reason for this invalidation is that the subclass of *Document* that is produced by the range of *sendsOrder* in the first triple is disjoint from the subclass of *Document* that is produced by the domain of *hasInvoiceNumber* in the second triple.

Theorem 6.3.3. ***Direct-Triple-Disjointness***
An annotation path $p = /t_1/.../t_n/$ is invalid, if there exist two logically valid neighbored triples $t_n = c_n/p_n/c_m$ and $t_m = c_m/p_m/c_{m+1}$, where $range(p_n) \sqcap restriction(c_n, p_n) \sqcap domain(p_m) \sqcap c_m \not\sqsubseteq Thing$.

Proof. The intersection class $range(p_n) \sqcap restriction(c_n, p_n) \sqcap domain(p_m) \sqcap c_m$ describes the implicit concept between two annotation triples, that is responsible for the concatenation of the triples. If this intersection concept is unsatisfiable any class with an existential restriction for this concept becomes unsatisfiable. □

6.3.3 Non-Local Invalidations

Local- and direct-triple-disjointness invalidations can be located precisely. That means the step in the path that causes the invalidation can be annotated with the type of the clash and the reason for the invalidation. This can be valuable information for a user who has to repair the annotation path. In case of general annotation paths which consist of two or more triples additional invalidations can occur which are not necessarily induced by neighboring triples. We will now first present an example in listing 6.3 and then define the problem in general.

```
1  Class(BusinessCustomer)
2  Class(Order), Class(BusinessOrder), Class(PrivateOrder)
3  Class(Itemlist)
4  Class(PrivateProduct), Class(BusinessProduct)
5  Class(Price)
6  Class(ClassMyAnnotation)
7  ObjectProperty(sends)
8  ObjectProperty(contains)
9  ObjectProperty(has)
10 ObjectProperty(hasPrice)
11 EquivalentClass(BusinessCustomer,
12     BusinessCustomer and sends only BusinessOrder)
13 EquivalentClass(BusinessOrder, BusinessOrder and has only
14     (Itemlist and contains only BusinessProduct))
15 EquivalentClass(MyAnnotation, Price and inv (hasPrice) some
16     (PrivateProduct and inv (contains) some
17         (Itemlist and inv (has) some
18             (Order and inv
19                 (sends) some BusinessCustomer))))
20 disjoint(BusinessOrder,PrivateOrder)
21 disjoint(BusinessProduct,PrivateProduct)
```

Listing 6.3: A non local invalidation

The example contains the annotation concept *MyAnnotation* that represents the path *BusinessCustomer/sends/Order/has/Itemlist/contains/PrivateProduct /hasPrice/Price*.

The annotation concept is free of local- or direct-triple-disjointness invalidations. Nevertheless, it is logically invalid because according to the ontology, business customers must only send business orders and business orders must only contain business products. Business products are disjoint from private products. This renders the whole annotation path unsatisfiable. Now our goal is to find the steps in the path that are responsible for the invalidation. In this case the steps that are responsible for the clash are:
BusinessCustomer/sends/Order/has/Itemlist/contains/PrivateProduct.

To define such invalidations we first introduce a normalized representation form of an annotation concept.

Definition 6.3.2. *Normalized Annotation Concept*
An annotation path $p = /c_1/p_2/c_3/...c_{n-2}/p_{n-1}/c_n/$ is represented as an annotation concept $p.c = c_n$ *and inv* (p_{n-1}) *some* $(c_{n-2}$ ... *and inv* $(p_2$ *some* $c_1)$ ...$)$. This annotation concept uses nested anonymous concepts. In contrast a normalized annotation concept of $p.c$ uses named concepts of the form:

$p.c = Ac_0 = c_n$ *and inv* (p_{n-1}) *some* Ac_1
$Ac_1 = c_{n-2}$ *and inv* (p_{n-3}) *some* Ac_2
$Ac_2 = c_{n-4}$ *and inv* (p_{n-5}) *some* Ac_3
...
$Ac_j = c_3$ *and inv* (p_2) *some* c_1

Definition 6.3.3. *Chain of Restrictions of an Annotation Path.*
A chain of restrictions of a normalized annotation concept $p.c$ of an annotation path p is any set of succeeding named concepts Ac_x .. Ac_{x+n} of $p.c$, where $n \geq 1 \wedge x \geq 0 \wedge x + n < |p.c| - 1$.

Theorem 6.3.4. ***Non Local Invalidations*** *When a path is invalid and it is free of local and direct-triple-disjointness invalidations, then there must exist at least one sub-path of two or more triples that conflicts with the ontology.*

Proof-Sketch: Given a logically invalid annotation path p_{inv} that is free of local- and direct-triple-disjointness invalidations of m triples of the form $/t1/../t_m$. From the absence of local invalidations follows that each triple $t \in p_{inv}$ is valid separately. From the absence of intra-triple-disjointness invalidations follows that the intermediate concept that is build by every neighbored pair of triples is satisfiable. Thus, the unsatisfiability of p_{inv} cannot have a local reason. Non-local invalidations are induced by chains of restrictions that conflict with the reference ontology. Each chain of restriction of an annotation path can also be represented as a sub-path of p_{inv}.

The discussed properties can be used to define and detect the minimal sub-path(s) of a logically invalid path that is responsible for the invalidation.

Definition 6.3.4. *Minimal Invalid Sub-path (MIS): An invalid sub-path p_s that is free of local or triple disjointness invalidations of an annotation path p is minimal, iff the removal of the first or last triple of p_s yields a satisfiable concept of p_s.concept in O.*

Before we proceed with an algorithm for the detection of *MIS* in section 6.3.4 we will now discuss which OWL constructs / patterns can cause non local invalidations.

Chain of Restrictions

There is is a chain of restrictions defined on concepts in O that produces a clash with the chain of restrictions of the MIS. One case of a chain of restrictions are nested concept definitions. We have already provided an example in listing 6.3. Of course such an invalidation can also be produced indirectly by named classes as shown in listing 6.4. In this case the annotation path *A/hasB/B/hasC/C/hasD/D* gets invalid because according to the ontology C may only have an inverse property *hasC* to N, and N may only have an inverse property *hasB* to E and E is disjoint from A, which is used in the annotation.
In the previous examples we have assumed that the concept that produces the clash is restricted with restrictions over inverse properties. In addition such a restriction chain can also be constructed in the reverse direction using non inverse properties. An example for such an invalidation is shown in listing 6.5. In this example the restriction on A requires A to have only a chain where the end of the chain is Z. Our annotation path ends with D, which is disjoint from Z. Thus, we would want to mark A and D to be responsible for the clash.

6

```
1  Class(A), Class(B), Class(C), Class(D), Class(E), Class(N)
2  Class(A/hasB/B/hasC/C/hasD/D)
3  ObjectProperty(hasB)
4  ObjectProperty(hasC)
5  ObjectProperty(hasD)
6  EquivalentClass(C, C and inv (hasC) only (N))
7  EquivalentClass(N, N and inv (hasB) only (E))
8  EquivalentClass(A/hasB/B/hasC/C/hasD/D, D and inv (hasD)
9      some (C and inv (hasC) some (B and inv hasB some A)))
10 disjoint(A,E)
```

Listing 6.4: Example ontology for an indirect restriction chain

```
1  Class(A), Class(B), Class(C), Class(D), Class(Z)
2  Class(A/hasB/B/hasC/C/hasD/D)
3  ObjectProperty(hasB)
4  ObjectProperty(hasC)
5  ObjectProperty(hasD)
6  EquivalentClass(A and hasB only (B and hasC only
7      (C and hasD only Z)
8  EquivalentClass(A/hasB/B/hasC/C/hasD/D, D and inv (hasD)
9      some (C and inv (hasC) some (B and inv hasB some A)))
10 disjoint(A,Z)
```

Listing 6.5: Example ontology for a non local invalidation with a non-inverse-chain

Of course such a clash can also be produced by using an indirect restriction chain using named concepts as shown in listing 6.6.

```
1  Class(A), Class(B), Class(C), Class(D), Class(N), Class(Z)
2  Class(A/hasB/B/hasC/C/hasD/D)
3  ObjectProperty(hasB)
4  ObjectProperty(hasC)
5  ObjectProperty(hasD)
6  EquivalentClass(A and hasB only N)
7  EquivalentClass(N and hasC only M)
8  EquivalentClass(M and hasD only Z)
9  EquivalentClass(A/hasB/B/hasC/C/hasD/D, D and inv (hasD)
10     some (C and inv (hasC) some (B and inv hasB some A)))
11 disjoint(A,Z)
```

Listing 6.6: Example ontology for a non local invalidation with an indirect non-inverse-chain

Transitive Properties

Transitive properties in a MIS can result in a clash, when some class in the *MIS* has a restriction that does not allow to have a property assertion to the inferred class. An example for such an invalidation is provided in listing 6.7.

```
1  Class(A), Class(B), Class(C), Class(D)
2  Class(A/hasB/B/hasC/C/hasD/D)
3  ObjectProperty(has)
4  transitive(has)
5  EquivalentClass(D and inv (has) only C)
6  EquivalentClass(A/hasB/B/hasC/C/hasD/D, D and inv (has)
7      some (C and inv (has) some (B and inv (has) some A)))
8  disjoint(C,B)
```

Listing 6.7: Example ontology for a MIS that is caused by transitive properties

The annotation concept of the annotation path $A/has/B/has/C/has/D$ gets unsatisfiable due to the transitivity of the property *has*. The transitivity has the consequence, that *D* gets an inverse property definition for *has* to *B* and *A* implicitly. Because *D* may only have an inverse relation *has* to *C* the annotation concept gets unsatisfiable.
This example also showed that even an annotation path that consists of only 2 triples can get invalid because of a non local clash. Of course we can also construct an example with a longer transitive chain, where the error in introduced at a triple that is not directly connected.

OWL Property Chains

OWL2 allows the definition of property chains which can also produce non-local clashes. A property chain has the following form *p1 o p2 –> p3*. It expresses that if there is a chain where some individual has a property $p1$ to another individual $i1$ and this individual has a property $p2$ to an individual $i2$, then the individual has the property $p3$ on $i1$. This can be seen as a more general case of transitivity, where the properties in the chain are not required to be in an equivalent- or sub-property relation in order to produce a MIS. An example for a MIS due to an OWL property chain is shown in listing 6.8.
In the example D may only have an inverse relation *hasChainedD* to X. The annotation concept gets unsatisfiable because it complies with the pattern of the property chain. Thus, A gets the property *hasChainedD* to D. This results in the inverse property *inv hasChained* on D for A which is forbidden by the definition of D.


```
1 Class(A), Class(B), Class(C), Class(D), Class(X),
2 Class(A/hasB/B/hasC/C/hasD/D)
3 ObjectProperty(hasB), ObjectProperty(hasC), ObjectProperty(hasD)
4 ObjectProperty(hasChainedD)
5 PropertyChain(hasB o hasC o hasD --> hasChainedD)
6 EquivalentClass(D and inv (hasChainedD) only (X)
7 EquivalentClass(A/hasB/B/hasC/C/hasD/D, D and inv (hasD)
8     some (C and inv (hasC) some (B and inv (hasB) some A)))
9 disjoint(A,B,C,D,E,X)
```

Listing 6.8: Example ontology for a non local invalidation by a property chain

Arbitrary Combinations

We can conclude that non-local invalidations of an annotation path can be constructed in OWL in various ways. Those are simple restriction chains that are defined with anonymous classes (as we define our annotation path expression) or by named classes. These expressions can be formulated in different directions. In addition to such restriction chains also transitive properties and property chains can lead to dependencies that span more than one concept. Finally those different constructs that can result in an unsatisfiable annotation concept can be mixed and the conflicting definitions can be inherited. We provide an example for an invalidation that uses multiple such patterns in listing 6.9. In the example the property chain *hasC,hasD –> hasCD* results in the additional *hasCD* assertion on B in the annotation path. Because there is a restriction on A that restricts the values of *hasB* to X and X has a restriction on *hasCD* that allows only Z and Z is disjoint from D that is derived from the property

chain the annotation concept gets unsatisfiable. Thus, we have a combination of a chain of restrictions and a property chain.

```
1  Class(A),Class(B),Class(C),Class(D),Class(X),Class(Y),Class(Z)
2  Class(A/hasB/B/hasC/C/hasD/D)
3  ObjectProperty(hasA), ObjectProperty(hasB)
4  ObjectProperty(hasC), ObjectProperty(hasD)
5  ObjectProperty(hasCD)
6  PropertyChain(hasC,hasD --> hasCD)
7  EquivalentClass(A/hasB/B/hasC/C/hasD/D, D and inv (hasD)
8      some (C and inv (hasC) some (B and inv (hasB) some A)))
9  EquivalentClass(X and hasCD only Z)
10 EquivalentClass(A and hasB only X and hasC only Y and
11     hasD only D)
12 EquivalentClass(D and inv (hasD) some (C and inv(hasC) some
13     (B and inv (hasB) some A)))
14 disjoint(Z,D)
```

Listing 6.9: Example ontology for a non local invalidations by a mixed chain

6.3.4 An Algorithm for the Detection of a Minimal Invalid Sub-Path

An algorithm for the detection of a minimal invalid sub-path of an annotation path p can be based on a structural search over conflicting axioms in the reference ontology. The last section has shown that such non local conflicts can occur due to many different OWL constructs. Of course a *MIS* can be the result of a combination of the described causes, which makes an exhaustive search even more complex. The efficiency of such an algorithm is further reduced by the fact that reasoning over sub-, super-, and equivalent-properties and -classes is required. In addition, for such a detection method the first and last triple of a sub-path are not known in advance. This makes another approach that directly operates on definition 6.3.4 of a minimal invalid sub-path more efficient. The corresponding algorithm is shown in in listing 6.10. The algorithm takes an invalid path p that is free of local and direct-triple disjointness invalidations as input and returns the index of the start and end triple of the detected *MIS*.
The algorithm uses some helper methods. The method *p.tripleCount()* returns the number of triples of p, *createSubPath(p,l,r)* returns the OWL expression of a sub-path of p that starts at index l and ends at index r. The methods assumes that the leftmost triple of p has the index 1 and the last triple of p has the index $p.tripleCount()$. The method *O.unsatisfiable(owlexp)* returns *true*, if the OWL expression *owlexp* is unsatisfiable in the ontology O.
The first loop is used to find the right boundary of a *MIS*. This is realized by sequentially creating a sub-path of p that begins at position 1 and end at position r, where r is decremented in each iteration. The loop terminates as soon as the created sub-path gets satisfiable. Therefore, the right boundary of the *MIS* must be at position $r + 1$. The reason for this is that analogues

to theorem 6.3.2, there can exist no complete *MIS* before position r. Otherwise r cannot be satisfiable. After the right boundary was found it is guaranteed that the sub-path between 1 and $r+1$ is invalid. However, it is not yet sure that it is minimal. Therefore, the left boundary of the *MIS* needs to be found. This is realized by creating a sub-path that begins at position l and ends at position $r+1$, where l starts at 2 and it is incremented in each iteration. As soon as such a sub-path gets satisfiable the left boundary of the *MIS* has been found at position $l-1$.

```
1  (int,int) getMinimalInvalidSubpath(p,O) {
2      r = p.tripleCount()-1;
3      // Find the right border of the MIS
4      while (O.unsatisfiable(createSubPath(p,1,r))
5              r--;
6          }
7      l = 2;
8      // Find the left border of the MIS
9      while (O.unsatisfiable(createSubPath(p,l,r+1))
10             l++;
11         }
12     return(l-1,r+1)
13 }
```

Listing 6.10: An algorithm for the detection of the minimal invalid sub-path

The detected *MIS* complies with definition 6.3.4 because both iterations guarantee that the removal of the first or last triple of the *MIS* result in a valid sub-path of p. The algorithm guarantees that is can find one *MIS*. If a path contains multiple *MIS*, we propose to remove them iteratively with the help of the proposed algorithm.

Theorem 6.3.5. *When the algorithm of listing 6.10 is used on a path that is free of local and direct-triple-disjointness invalidations that contains multiple MIS, then the leftmost inner MIS is detected.*

Proof-Sketch: Given an annotation path $p = /t_1/t_2/.../t_n$. In the first iteration sub-paths starting at t_1 of p are created. The unsatisfiable sub-path with the minimum number of triples is considered to be a *MIS*-candidate. According to theorem 6.3.2 there can exist no other complete *MIS* in the path that ends before the *MIS* candidate. It is only possible that there exists another *MIS* that starts before and ends after or at the same position as the detected one. In the next loop the minimality of the *MIS* is guaranteed by chopping elements from the start. As a consequence the algorithm detects the leftmost inner-*MIS*.

6.4 Implementation Considerations

In this chapter we have defined error-types on annotation paths. The goal is to tell the user, which steps of a path are responsible for the invalidation including an explanation of the type of invalidation. The detection of most invalidation types is straight forward. It is just a query to the reasoner that is equivalent to the definition of the specific invalidation type. Non local invalidations can be tracked by the proposed MIS algorithm. However, testing each triple of every invalid annotation path can be an expensive task. Therefore, we will briefly discuss properties of annotation path that can be used to enormously reduce the number of queries to the reasoner. In a typical scenario there is a set of valid annotations V and a set of invalid annotations I. Both sets are a direct result of the classification of the reference ontology with the added annotation concepts. We now define properties that hold between the elements of V and I.

Theorem 6.4.1. *Globally-valid path-postfix:* Given a set of valid annotations V and one invalid annotation i. If there exists an annotation v in V with a common postfix (ending with the same sequence of triples) with i, then the corresponding sub-path of i cannot introduce local or direct-triple-disjointness invalidations.

Proof. No annotation path $\in V$ can contain local or direct-triple-disjointness invalidations. Otherwise it would not be satisfiable. If a path is satisfiable also every postfix of it must be satisfiable due to the monotonicity of OWL. An annotation concept is a specialization of the last concept-step. The longer a path is, the more specific is the annotation concept. When there exists an annotation path $v \in V$ which has the same postfix f as i, then $i.concept$ is a more specific concept than $f.concept$. Thus, the additional specialization must induce the error. It is represented by the prefix of i which does not match f. □

Theorem 6.4.2. *Globally-valid-triples:* A triple that is an element of a path $\in V$ cannot produce a local invalidation when it is used in a path in I.

Proof. The proof of theorem 6.4.2 is a direct consequence of theorem 6.3.2. Any triple that is an element of a valid path cannot be logically invalid because otherwise the path would be invalid. □

These two properties of a globally valid postfix and triples can be used to find local invalidations or intra-triple-disjointness invalidations very efficiently. As a first step the longest common postfix from i and the annotations in V can be detected. If such a postfix is found it is guaranteed that the corresponding postfix in i cannot contain local or direct-triple disjointness invalidations. In addition all triples in all paths of V can be considered as locally valid triples. Thus, if they occur in i they do not need to be tested for local invalidations.
Finally, when the annotation concepts are represented in form of normalized concepts (see definition 6.3.2) in the ontology, it is guaranteed that all triples that correspond to satisfiable

named concepts are locally valid and that no direct-triple-disjointness invalidations can exist between succeeding triples, that correspond to satisfiable named concepts in the normalized representation.
All these considerations can lead to a major speedup for the detection of invalidations because triples and combinations of triples that are known to be valid do not need to be checked for specific error-types (domain-invalidation, range-invalidation, ...) and in order to guarantee that the input path of the *MIS* algorithm is free of local or direct triple-disjointness invalidations, only the potentially invalid triples and combinations of triples need to be checked.

6.5 Additional Justifications of Invalidations in Annotation Paths

So far we have defined invalidation types and provided algorithms and methods to efficiently track invalidations. This information can be used to find a logically valid variation of the annotation path. While local and direct-triple-disjointness invalidations can be located exactly, non local invalidations can only be tracked in form of a minimal invalid sub-path (MIS). If the user has difficulties in finding the reason for the invalidation based on a *MIS*, general debugging methods to justify invalidations can be used additionally. The computation of justifications or the minimal subset of axioms that are responsible for the unsatisfiability of a concept is studied in the field of ontology debugging (see section 6.6). We distinguish two different cases for the generation of such additional justifications for the invalidation: General justifications, when the annotations are introduced for the first time and the detection of changes that are responsible for invalidations after ontology evolution.

6.5.1 General Justifications

Typical approaches to justify/explain, why a concept is unsatisfiable are based on the calculation of a Minimal Unsatisfiability Preserving sub-Tbox (MUPS) [89] of an unsatisfiable concept. A Black-Box algorithm to detect one MUPS is presented in [45]. It is directly inline with our assumption of a reasoner-independent annotation debugging system. The calculation of the MUPS is straight forward and is shown in figure 6.2.

The algorithm begins with an empty Tbox and adds sets of axioms until the concept in question becomes unsatisfiable. In a next step the generated subTbox needs to get minimal. Thus, the removal of any element must render the concept satisfiable. This is realized by a subsequent removal of axioms. After each removal step the concept is tested for satisfiability. If it is satisfiable then the axiom cannot be removed otherwise it is removed. Between the addition and the subsequent deletion of axioms is a fast-pruning step. It is realized by a sliding window. Thus, instead of removing single axioms a set of axioms is removed at once.

Algorithm: SINGLE_JUST_ALG$_{Black-Box}$
Input: Ontology $\mathcal{O}$, Unsatisfiable concept C
Output: Ontology $\mathcal{O}$'
(1) $\mathcal{O}' \leftarrow \emptyset$
(2)**while** (C is satisfiable w.r.t $\mathcal{O}$')
(3) select a set of axioms $s \subseteq \mathcal{O}/\mathcal{O}'$
(4) $\mathcal{O}' \leftarrow \mathcal{O}' \cup s$
(5)perform fast pruning of $\mathcal{O}$' using a sliding window technique
(6)**for each** axiom $k' \in \mathcal{O}'$
(7) $\mathcal{O}' \leftarrow \mathcal{O}' - \{k'\}$
(8) **if** (C is satisfiable w.r.t. $\mathcal{O}'$)
(9) $\mathcal{O}' \leftarrow \mathcal{O}' \cup \{k'\}$

Figure 6.2: The black-box MUPS algorithm from [45]

This reduces the number of reasoner calls and is responsible for a major performance gain. An important tuning parameter is the selection of the set of axioms that is added during the first step. The authors of [45] describe that in their implementation they slowly start to add more and more axioms beginning at the concept in question. Then they add axioms that are structurally connected to the axioms of the concept.
While this is a good general approach for arbitrary debugging scenarios we propose that our findings about non local invalidations can be used as a heuristics to further tune this approach for our application scenario:

- Add the axioms of the annotation concepts between the beginning and the end of the minimal invalid sub-path p_m including definitions of the property-characteristics and property chains.
- Perform a structural search for a path between the beginning and the end of p_m in O.
- Add the axioms of the elements that where found, during the structural search.
- Subsequently add axioms that are connected to the found elements/axioms.

The structural search can be implemented as an adoption of the proposed algorithm in [77]. It should exploit the presented patterns for non local invalidations from section 6.3.3. Since the corresponding properties and the start and end of the minimal invalid sub-path are known, we can efficiently search for such patterns.
There can potentially be multiple MUPSs for one unsatisfiable concept. Thus, when all MUPSs of an unsatisfiable concept are found and at least one axiom of each MUPS is removed the concept becomes satisfiable. Algorithms to find all MUPSs based on one MUPS are provided in [45]. Since this thesis focuses on the maintenance after ontology evolution we will not go into detail for this potential heuristic optimization for the generation of justifications.

6.5.2 Justifications after Ontology Evolution

In case of annotations maintenance after ontology evolution we assume that the now unsatisfiable annotation concept was satisfiable in the old ontology version. Therefore, it must be possible to find the reason for the invalidation in a change-LOG that captures the changes between the old ontology version O_n and the new version O_{n+1}. As discussed in chapter 4 an appropriate LOG of changes for OWL ontologies is a *LOG* of added and deleted axioms.

This leads to the following problem: Given a *MIS*, *mis* of an unsatisfiable annotation concept in O_{n+1} and a change-LOG C that, when applied on O_n produces O_{n+1}, find the minimal set of axioms of C that is responsible for the invalidation of *mis*. C consists of two sets of axioms: $C.deleted$ and $C.added$.

6

Theorem 6.5.1. *The justification for the invalidation of a MIS must be a subset of* $C.added$*.*

Proof. This is a direct consequence of the monotonicity of OWL. A deleted axiom $\in C.detelted$ can never render a concept logically invalid. Therefore, the set of conflicting axioms from the LOG can only be a subset of $C.added$. □

After having shown that we must find the set of axioms that are responsible for the invalidation in $C.added$, we only need to find this set of axioms to justify the invalidation. This can be realized with an adopted MUPS algorithm that basically does the following:

- Create a new ontology O', where $O' = O_n \setminus C.deleted$.
- Subsequently add axioms of $C.added$ to O' until *mis* gets unsatisfiable in O'.
- Subsequently remove the added axioms to proof that *cmin* is minimal.

The result of this algorithm is a minimal set of axioms of the change-LOG *cmin* that is responsible for the invalidation of the *mis*. We suppose that this is already valuable information for the user who needs to adopt the annotation path in order to comply with the new ontology version. Because typically only a small set of axioms is modified between two ontology versions, we can assume that the computation of *cmin* is far less expensive that the computation of a MUPS.
The set of axioms *cmin* can additionally be used to find related change-descriptions of the change ontology that is described in section 4.5. Each found axiom must have a relation to some named entity in the ontology. Thus, we can query all changes on the related entities (including sub/super- class and property relations) from the change ontology and present them to the user in addition to the found conflicting axioms.

6.6 Related Work

The annotation method of this research has two representations: path expressions and complex OWL formulas. Only the path expressions can be changed by the annotators. Therefore, we have proposed methods to track errors in annotation paths. In order to find errors in the corresponding complex OWL formula also general ontology debugging solutions could be used. However, preliminary experiments with the well known OWL1 tool Swoop [77] have shown, that Swoop was not able to detect the root-cause of many non local invalidations that only used OWL1 language constructs. In this case the concept was detected to be invalid but no explanation could be generated. When explanations could be generated it was very tedious for the annotator to actually discover which elements in the path were responsible for the problem. The integrated repair tool of Swoop could not help either. In contrast our method can precisely track which elements in the path are responsible for the invalidation and it is a reasoner-independent black-box approach. In addition we have defined error-types that indicate the reason for the invalidation with respect to the steps of the path.
A fundamental publication in the field of ontology debugging is [89]. It introduces the term of minimal unsatisfiable sub Tboxes (MUPS). A MUPS is a minmal set of axioms that is responsible for a concept to be unsatisfiable. When one axiom gets removed from the MUPS the concept gets satisfiable unless, there are additional MUPS for the concept. This definition is somehow analogues to our definition of the minimal invalid sub-path. In [45] an optimized black box algorithm for the computation of the MUPS is presented. The Black-Box algorithm basically tries to find the MUPS in a trial and error fashion, which requires a high number of expensive reclassifications. In order to get all justifications the authors calculate a first justification (MUPS) and afterwards use a variant of the Hitting Set Algorithm [86] to obtain all other justifications for the unsatisfiability.
The goal of general ontology debugging approaches is: Given an ontology with at least one unsatisfiable concept find a set of axioms that need to be removed in order to obtain a coherent ontology. There can be multiple sets of such axioms (also called diagnoses). Therefore, it is beneficial to rank the possible repairs either by assuming that the set of removed axioms should be minimal [89] or by selecting the diagnosis [44] that best fits the modeling intention by asking an oracle/user. This is a major difference to the annotation maintenance scenario, where the ontology cannot be changed by the annotator and only changes of the the path expression are allowed. Therefore, we search especially for steps in the path that lead to an error.
An alternative approach to debug ontologies are patterns/anti-patterns (in particular logically detectable anti-patterns) as proposed in [17, 16]. Those patterns concentrate on common modeling errors that are made on ontology artifacts such as concepts. They can provide well understandable explanations for common errors on simple concepts. Because the subject of such patterns is a concept and not an annotation path their usefulness for annotation paths is limited to simple cases.

6.7 Conclusion

In this chapter we have addressed the problem of the logical invalidation of annotation path expressions. A logical invalid annotation path expression is a structurally valid path expression, where the corresponding annotation concept is unsatisfiable in the reference ontology. We have provided an in depth analysis of the possible causes for logically invalid annotations. There are basically two types of invalidations: local invalidations and non-local invalidations. We have provided efficient detection methods for both types of invalidations. We expect that experts who annotate artifacts will be much more efficient, if the position and the cause of errors in annotations paths is automatically determined. This technique is also particulary useful for annotation maintenance as a consequence of an evolution of the reference ontology. Our method not only identifies the annotations which became logically invalid, but also narrows the inspection area to the shortest possible path and gives indications on the causes of the invalidation. For non local invalidations is can be useful to provide additional justifications for invalidations. We have provided two approaches to generate such justifications: One approach for the generation of justifications of new annotations and one approach specifically for the maintenance of annotations. All proposed algorithms are built upon the functionality usually provided by generic reasoners for OWL ontologies, so they are not restricted to a specific reasoner or ontology management system.

Chapter 7

Detection of Semantic Changes

[1]We have already discussed the problem of structural and logical invalidations of annotation paths in chapter 5 and 6. An annotation path is structurally valid, when the structure of the path complies with the restrictions of the annotation method. The structural validity is a precondition to transform an annotation path to an OWL concept. This is, where the logical validation comes into play. An annotation concept that is not satisfiable in the reference ontology is a logically invalid annotation path.
Nevertheless, changes in the reference ontology can have additional consequences for the annotations and especially for the interpretation of instance data. Such changes may require to change the instance data in order to comply with the new ontology version. We call this kind of invalidation semantic invalidation. In this chapter we will investigate, how semantic invalidation can be tracked.

We will first present the notion of semantic changes and discuss if such changes to the semantics of an annotation can be derived from the plain ontology in section 7.1 and describe and define explicit dependency definitions in section 7.2 and 7.3. In section 7.4 we show how the explicit definitions can be used to track semantic changes. In section 7.5 we present a prototype-implementation of the approach. Section 7.6 gives an overview of the related work.

7.1 Semantic Changes and their Automatic Detection

Semantic changes are consequences of changes in the reference ontology which do not invalidate the annotation structurally or logically, but might lead to misinterpretations. We will illustrate such semantic changes with the following example:

[1]The content of this chapter has been published in [58].

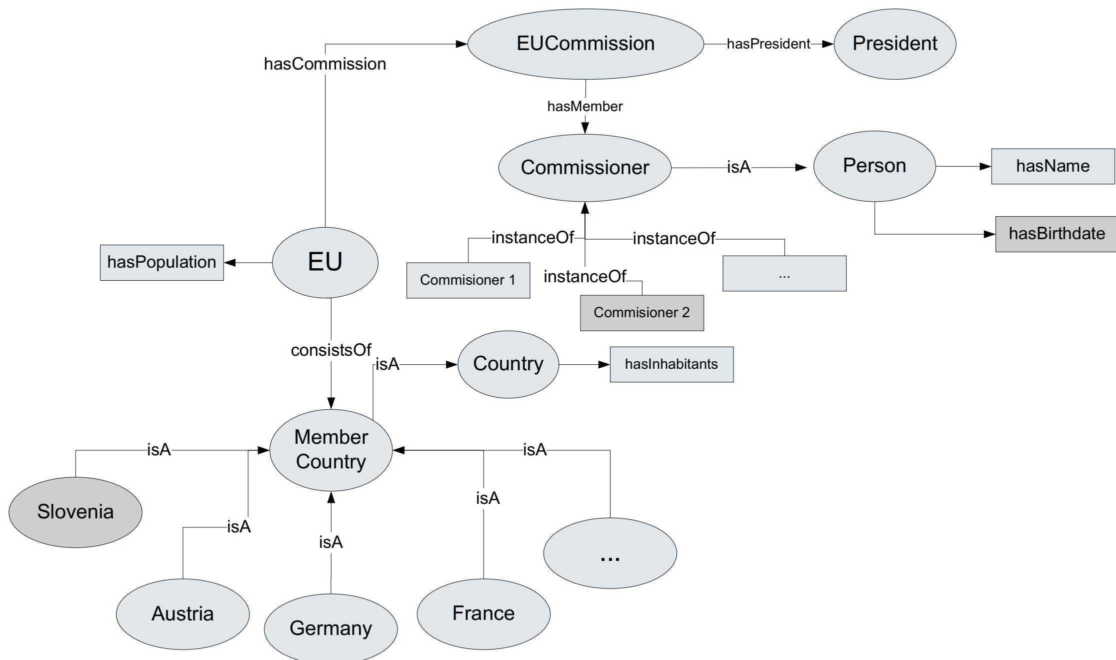

Figure 7.1: Example ontology

In figure 7.1 an example ontology[2] that represents parts of the European Union is depicted. We now assume that we have an annotation of an XML-Schema-element with the annotation path */EU/hasPopulation*. Some changes were made to the ontology: *Slovenia* was added as an additional *MemberCountry* and an additional *Commissioner*-instance was added and persons now have the property *hasBirthdate*. The differences between the old and the new ontology version are marked in dark-grey. Now obviously the semantics of an annotation */EU/hasPopulation* of some element in an XML Schema are changed because the parts of the European union were changed. This change does not influence the validity of the annotation itself - but the semantics of the annotation has changed. Documents that were annotated with the old ontology version will have a lower population number than documents of the current version. This imposes problems because it leads to the misinterpretation of the data. For example a human reader might come to the conclusion that the EU has a higher birthrate after the change. The goal of this chapter is the automatic generation of warnings about such changes.
Since an ontology is used to express the semantics of a domain it should be possible to derive the changes of the semantics of annotations automatically. To avoid that each annotation has to be checked, if any ontology element has changed, we need to reduce the set of ontology elements which might invalidate a specific annotation. Ontology views [73] are methods to reduce the size of an ontology. Ontology module extraction [18] techniques can be applied

[2]The member countries are modeled in form of concepts and specific commissioners as instances in order to show problems that may occur when concepts or instances evolve.

for the same purpose. These methods generate a sub-ontology that only contains relevant elements for a given starting point (set of concepts). The starting point in our case is the semantic representation of an annotation path expression. When such sub-ontologies are created for the old and the new ontology version we can check, if there were changes between the sub-ontology versions. Since the sub-ontologies only contain elements that are relevant for the annotation in question we should be able to significantly reduce the number of false-positives. Typical methods for the generation of sub-ontologies begin with a concept in question and then add more and more concepts that are related. The relation is expressed in form of sub/-superclass relations or object-properties. We will illustrate the general idea of the generation of a sub-ontology with an example:

- The starting point is *EU/hasPopulation*
- This leads the inclusion of the concept *EU*
- The *consistsOf* property on *EU* requires the addition of *MemberCountry*
- *MemberCountry* requires the addition of its superconcept: *Country* with the property *hasInhabitants*
- *MemberCountry* requires the addition of *Slovenia*, *Austria*, *Germany*, ...

When we now compare the view of the old and the new ontology version we can figure out that *Slovenia* was added. We would throw a warning that the semantics of *EU/hasPopulation* was possibly changed. In this example we have assumed that we include all datatype-properties of a concept in the view and all concepts that are in the range of the object properties of the included concepts. In addition, all super- and subconcepts as well as individuals of the included concepts are added. Unfortunately, such an algorithm would include much more concepts:

- The *hasCommission* property of *EU* requires the addition of *EUCommission*
- The *hasPresident* property of *EUCommision* requires the addition of *President*.
- The *hasMember* property of *EUCommision* requires the addition of *Commissioner*.
- The *Commissioner* concept requires the addition of its instances.
- ....

At the end the entire ontology would be included in the view. Thus, all changes that happened between the old and the new ontology version are assumed to be relevant for *EU/hasPopulation*. This is certainly not true. In order to avoid this behavior the set of properties that are followed to build the view needs to be much smaller. Thus, strategies are required to choose the proper object-properties that should be followed. But where is the difference between *consistsOf* and *hasCommission*? From where can we know that if we want to build the view for *EU/hasPopulation* we need to follow the *consistsOf* property and that if we want to create the view for

EU/numberOfCommissioners we need to follow the *hasCommision* property?
Thus, strategies are required in order to keep the view small and meaningful.

7.2 Requirements for Explicit Dependency-Definitions

As shown in the last section there is typically no knowledge about what may be invalidated semantically by changes since this is not an invalidation of the logical theory (which could be calculated) but a change of the semantics of the annotations. The reason for this is that the ontology does not fully specify the real-world domain. Therefore, a straight forward solution is the addition of the missing knowledge to the ontology. This means sentences like "*The population of the EU is changed when the Member Countries change*" should be added to the ontology. Obviously this a very wide definition because we have not stated anything about the types of relevant changes. Is it changed when the name of a country changes or only if a specific attribute changes? In general which operations may invalidate our value? The examples of the population of the EU can be described as the aggregation of the population of the member countries. Thus, we need a way to describe such functions. These observations lead to the following requirements for change-dependency definitions:

1. The change-dependent concept or property must be described including the context. For example *hasPopulation* of *EU* and *hasPopulation* of *City* might depend on a different set of ontology elements.
2. The definition of the change-dependency should allow fine grained definitions of dependencies. For example it should be possible to define that *EU/hasPopulation* is dependent on the *population* of the *Member Countries*. It would not be sufficient to state that it is dependent on *population* in general.
3. It should be possible to define that one artifact is dependent on a set of other artifacts.
4. Multiple dependencies should be possible for one change-dependent concept or property.

According to the first requirement the dependent artifact needs to be specified precisely. This can be realized with the annotation path syntax. Therefore, the path *EU/hasPopulation* defines that the *hasPopulation* property on the concept *EU* is the subject of a dependency definition. The second requirement supposes that not only the subject should be described via path expressions but also the object of a change-definition should be described in terms of path expressions. Unfortunately, the plain annotation path syntax does not fulfill the third requirement. Therefore, it needs to be enhanced with expressions to address sets.

Dependencies on Sets and Aggregations: Some ontology artifact may be change-dependent on a set of other artifacts. In our running example the population of the *EU* depends on the set of *Member Countries*. More precisely it is not dependent on the set of *Member Countries* in

general but on the sum of the *hasInhabitants* property of each *MemberCountry*. In general, there are different kinds of sets in an ontology: subclasses, sub-properties and instances. Therefore, all those must be expressible. The sum function is only one aggregation function. Typical other aggregation functions are min, max, count, and avg. In addition to aggregation functions another kind of function over sets is of interest: The value function. It can be used to state that one artifact is directly dependent on the values of a set of other artifacts. The subclasses and instances operator can be used in a path wherever a concept is allowed and the sub-properties operator can be used wherever a property-step is allowed. We will illustrate the ideas with examples:

1. *EU/hasPopulation* is dependent on */EU/consistsOf/sum(subclasses(MemberCountry))/hasInhabitants*.
2. *EU/numberOfCommissioners* is dependent on */EU/hasCommission/ EUCommission/hasMember/count(instances(Commissioner))*.
3. *MemberCountry/hasInhabitants* is dependent on *MemberCountry/subproperties(hasInhabitants)*.
4. */city* is dependent on *value(subclasses(city))*

The first example calculates the sum of all *hasInhabitants* properties of all subclasses of *member – countries*, while the second one just counts the number of *commissioner* instances. The first example defines an abstract sum because the ontology cannot contain any information about the number of inhabitants on class level. It only defines that the value becomes invalid if the ontology structure changes in a way that the function would operate on a changed-set of ontology artifacts. In contrast the second example can return a defined number because it is a simple count operation. In addition, it defines the change-dependency over instances. In this case the ontology may contain instance data. In the third example it is assumed that the *hasInhabitants* property has sub-properties and that a change of the sub-properties will also invalidate *EU/hasPopulation*. Examples for sub-properties could be *hasMalePopulation* and *hasFemalePopulation*.
The last example shows the value function. It defines that elements that are annotated with */city* are change-dependent on all the subclasses of *city*. Therefore, a rename of a subclass of *city* requires a rename of the specific city-element in XML-documents as well.

7.3 Definition of Change-Dependencies

In order to introduce the proposed change-dependency definitions we will first briefly recall our ontology model. We use the abstract ontology model from chapter 4 it is shown graphically in figure 4.4. This ontology model covers the important concepts of most ontology languages. By using this abstract model we can apply the work on different ontology formalisms that can be transformed to our representation. This does not require to transform the

whole ontologies to our ontology model. It is sufficient to formulate the changes that occur to the ontology in terms of our ontology-model. Depending on the used ontology formalism reasoning may induce additional changes that we can simply also add to our change-LOG by comparing the materialized ontology version before and after the change.

Based on this ontology model we define annotation paths that are used to annotate artifacts of an XML-Schema with a reference ontology. Both annotation path and dependency-definitions are modeled in the meta-model in figure 7.2. An *AnnotationPath* consists of a sequence of *AnnotationPathSteps*. Each step has a position and a *uri* that points to some property or concept of the reference ontology. According to the referenced ontology artifact an *AnnotationPathStep* is either a *conceptStep* or a *propertyStep* with the subclasses *objectPropertyStep* and *dataTypepropertyStep*. Each step except the last step has a succeeding step. Each step except the first step has a previous step. An annotation path has a defined first and a defined last-step.
A *DependencyDefinition* has one *hasSubject* relation to an *AnnotationPath* and one or more *hasObject* relations to *dependencyDefinitionPath*. Each *dependencyDefinitionPath* consists of a number of *DependencyPathSteps*. A *DependencyPathStep* is a subclass of an *annotationPathStep* which is extended with an optional *setExpression*. The *setExpression* has a *type (subclasses, subproperties, instances)* and an optional *function* which has a type that can be *value*, *min*, *max*, *avg*, *count*. Each *DependencyDefinitionPath* has a *hasAnnotationPath* relation to one *AnnotationPath*. This specific *annotationPath* is created by casting all steps to standard *AnnotationPathSteps*. An *annotationPath* can be represented in form of an ontology concept. This concept can be obtained with the method *getConcept*().
In order to meet the constraints of the annotation method some integrity constraints on *AnnotationPath* and *DependencyDefinitionpath* are required.

7.3.1 Integrity Constraints on Annotation Path

1. The first step must be a *ConceptStep*.
2. An *AnnotationPath* must not contain *DependencyPathSteps*.
3. The last step must be a *ConceptStep* or a *dataTypePropertyStep*.
4. When a *conceptStep* has a previous step then the previous step must be an *ObjectPropertyStep*.
5. The next step of a *ConceptStep* must be an *ObjectPropertyStep* or a *DataTypePropertyStep*.
6. A *DataTypePropertyStep* can only exist as the last-step.
7. A *ConceptStep* must not reference to another *AnnotationPath*.

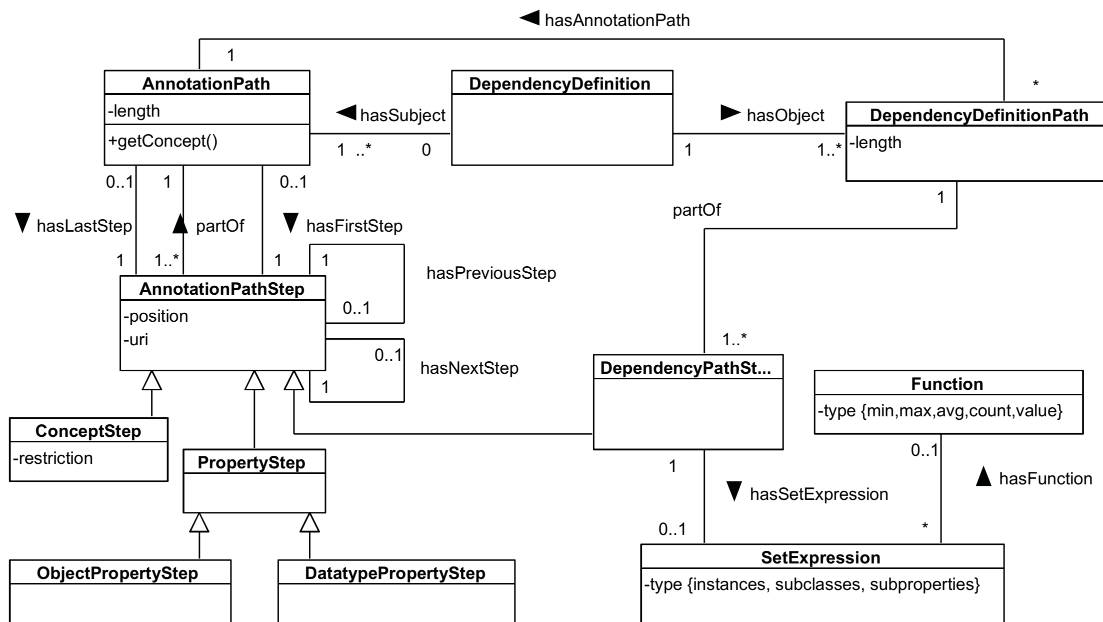

Figure 7.2: Meta-model of the change-dependency definitions

7.3.2 Integrity Constraints on Dependency Definition Path

1. All integrity constraints of standard steps except (2) and (7) also apply on *DependencyDefinitionPath*.
2. Only the last two steps may have a *setExpression* including a *function*.
3. The *setExpression* of type *subclasses* is only allowed for *conceptSteps*.
4. The *setExpression* of type *instances* is only allowed for *conceptSteps*.
5. The *setExpression* of type *subproperties* is only allowed for *propertySteps*.

7.4 Detection of Semantic Changes

Given a set of annotations, a representation of changes analogues to chapter 4 and a set of dependency definitions we need to find annotation paths that are possibly semantically invalid. In addition we need a report, why each possibly invalid path got invalid. The change representation stores the changes in form of instances of a change ontology. The detection of relevant changes is realized by rules that operate on the statements of the change representation and the old and the new ontology versions.

We now define semantic invalidation as a change affecting the semantics of an annotation path as follows. The predicate $matches(a, b, O_n)$ returns *true*, iff the annotation path a matches the annotation path b with a strong match (equivalence- or subclass- relation) with regard to the ontology version O_n (see section 3.2.2). The predicate $subClassOf(subc, superc, O_n)$ states that $subc$ is an equivalent or subclass of the superclass $superc$ according to the ontology version O_n. The predicate $subPropertyOf(subp, superp, O_n)$ expresses the sub-property-relationship analogously.

Definition 6. ***Semantic Invalidation of an Annotation-Path:***

Given an ontology version O_n, a succeeding ontology version O_{n+1}, a set of changes $Changes(O_n, O_{n+1})$ abbreviated by C, a set of explicit dependency-definitions DEP, and a set of XML-Schema-annotations A. An annotation path $a \in A$ is semantically invalid if:

$semInvalid(a, C, DEP, O_n, O_{n+1}) \leftarrow InvalidByDep \neq \{\}$

$RelevantDependencies$ is the set of definitions where the corresponding subject is an equivalent- or superclass of the annotation path a:
$RelevantDependencies = \{\forall dep \in DEP | matches(a, dep.subject, O_n)\}$
$InvalidByDep$ is the set of change-dependency definitions where one of the objects got invalid because it contains a step that is invalid or if the semantics of the annotation path of the object itself got changed.

$InvalidByDep = \{\forall dep \in RelevantDependencies | (hasObject(dep, obj)$
$\wedge isInvalid(obj)) \vee (hasAnnotationPath(obj, annotationPathObject)$
$\wedge semInvalid(annotationPathObject, C, DEP, O_n, O_{n+1}))\}$

Thus, dependency-definitions are transitive. If a depends on b and b depends on c then a is invalid when c is invalid.

A $DependencyDefinitionPath$ is invalid if at least one of its steps is invalid:
$isInvalid(obj) \leftarrow \exists step \in obj.steps \wedge InvalidStep(step)$
When a step is invalid is described in the next subsections.

Rules for the Invalidation of Steps:

For the sake of simplicity we will define the invalidation of steps in form of rules omitting quantifiers. In addition all rules operate on the change-set defined by $Changes(O_n, O_{n+1})$. The rules 1-3, 6, 8, 10 create possible invalidations, while the others create invalidations. Possible invalidations are invalidations where an invalidation may have taken place but additional review by the user is required.

1. A *PropertyStep* gets possibly invalid, if the domain of the property or of a super-property has changed.
$PropertyStep(?step) \wedge subPropertyOf(?step.uri, ?superProperty,$
$O_{n+1}) \wedge updateDomain(_, ?superProperty, _, _)$
$\Rightarrow InvalidStep(?step, 'DomainOfPropertyChanged')$

2. A property-step is possibly invalid, if the range of the property or a super-property has changed.
$ObjectTypePropertyStep(?step) \wedge subPropertyOf(?step.uri, ?superProperty,$
$O_{n+1}) \wedge updateRange(_, ?superProperty, _, _)$
$\Rightarrow InvalidStep(?step, 'RangeOfPropertyChanged')$
The same holds for the change of the data type of a datatype-property analogously.

3. A concept-step gets possibly invalid, if a restriction on the property of the next step has changed.
$ConceptStep(?step) \wedge isSubConceptOf(?step.uri, ?superuri, O_{n+1}) \wedge$
$hasNextStep(?step, ?next) \wedge subPropertyOf(?next.uri, ?supernexturi, O_{n+1})$
$\wedge updateRestriction(_, ?superuri, ?supernexturi, _, _, _, _)$
$\Rightarrow InvalidStep(?step, 'RestrictionOnNexStepChanged')$

4. A set expression over subclasses without a function becomes invalid, if a subclass is added.
$ConceptStep(?step) \wedge hasSetExpression(?step, ?exp) \wedge$
$!hasFunction(?exp, _) \wedge equals(?exp.type, 'subclasses') \wedge$
$subConceptOf(?suburi, ?step.uri, O_{n+1}) \wedge addChildC(_, ?newc, ?suburi)$
$\Rightarrow Invalid(?step, 'SubclassAdded')$

5. A set expression over subclasses without a function becomes invalid, if a subclass is removed.
$ConceptStep(?step) \wedge hasSetExpression(?step, ?exp) \wedge$
$!hasFunction(?exp, _) \wedge equals(?exp.type, 'subclasses') \wedge$
$subConceptOf(?suburi, ?step.uri, O_{n+1}) \wedge remChildC(_, ?newc, ?suburi)$
$\Rightarrow Invalid(?step, 'SubclassRemoved')$

6. A set expression over subclasses without a function becomes possibly invalid, if a restriction on the property of the next step is changed in one of the subconcepts.
$ConceptStep(?step) \wedge hasSetExpression(?step, ?exp) \wedge$
$!hasFunction(?exp, _) \wedge equals(?exp.type, 'subclasses')$
$\wedge subConceptOf(?suburi, ?step.uri, O_{n+1}) \wedge hasNextStep(?step, ?next)$

$\wedge\, subPropertyOf(?nextpropuri, ?next.uri, O_{n+1}) \wedge$
$updateRestriction(_, ?suburi, ?nextpropuri, _, _, _, _)$
$\Rightarrow Invalid(?step.'RestrictionOnSubclassChanged')$

7. A set expression over instances without a function becomes invalid, if instances are added or removed to/from the specified concept or one of its subconcepts. We will only depict the rule for the addition here.
   $ConceptStep(?step) \wedge hasSetExpression(?step, ?exp) \wedge$
   $!hasFunction(?exp, _) \wedge equals(?exp.type, 'instances') \wedge$
   $subConceptOf(?conceptUri, ?step.uri, O_{n+1})$
   $\wedge\, addInstToC(_, _, ?conceptUri) \Rightarrow Invalid(?step, 'InstancedAdded')$

8. A set expression over instances becomes possibly invalid, if the succeeding-step is a property-step and property assertions on instances for that property are modified.
   $ConceptStep(?step) \wedge hasSetExpression(?step, ?exp) \wedge$
   $!hasFunction(?exp, _) \wedge equals(?exp.type, 'instances') \wedge$
   $subConceptOf(?conceptUri, ?step.uri, O_{n+1}) \wedge hasNextStep(?step, ?next) \wedge$
   $PropertyStep(?next) \wedge instanceOf(?insturi, ?conceptUri) \wedge$
   $updatePropertyAssertion(_, ?insturi, ?next.uri, _, _)$
   $\Rightarrow Invalid(?step, 'PropertyAssertionChanged')$

9. A set expression over sub-properties becomes invalid, if a sub-property is added or removed. We will only depict the rule for the addition here.
   $PropertyStep(?step) \wedge hasSetExpression(?step, ?exp) \wedge$
   $!hasFunction(?exp, _) \wedge equals(?exp.type, 'subproperties') \wedge$
   $subPropertyOf(?suburi, ?step.uri, O_{n+1})$
   $\wedge\, (addChildOProp(_, ?newc, ?suburi) \vee (addChildDProp(_, ?newc, ?suburi))$
   $\Rightarrow Invalid(?step, 'SubpropertyAdded')$

10. A set expression over sub-properties becomes possibly invalid, if the domain or range of a sub-property is changed. *uDomainOrRange* is the superclass of *updateDomain* and *updateRange*
    $PropertyStep(?step) \wedge hasSetExpression(?step, ?exp) \wedge$
    $!hasFunction(?exp, _) \wedge equals(?exp.type, 'subproperties')$
    $\wedge\, subPropertyOf(?suburi, ?step.uri, O_{n+1}) \wedge$
    $uDomainOrRange(_, ?suburi, _, _)$
    $\Rightarrow Invalid(?step, 'DomainOrRangeOfSubpropertyChanged')$

Functions on Set Expressions:

The rules in the last section excluded the existence of functions over *setExpressions*. Therefore, any change that has consequences for the *setExpression* is considered to invalidate the step. When a function is given then the problematic change-operations depend on the used function and, therefore, additional rules are required. The sum-function is vulnerable to *add* and *delete* operations but is resistent to local *merge* or *split* operations. All other aggregation functions are vulnerable to *add*, *del*, *split*, and *merge*. The value function is vulnerable to renames of subconcepts or sub-properties as well as to *delete*, *split* and *merge* operations. Therefore, specific rules for the different kinds of functions are required. Since *merge* and *split* are composite change-operations the rules need to operate on the annotation of the changes (*ChangeAnnotation*(...)). We will provide the rules for sum-functions with added subconcepts and value-functions with renames here.

1. A concept-step with a sum-function over subconcepts gets invalid, if subconcepts are added or removed and the add and remove operations are not linked to local split or merge operations. A non-local split operation happens when the source concept was a subconcept of the step and one of the new concepts is not, still, a subconcept of the step according to the current ontology version. The following rule represents the case of the addition of subconcepts.

   $ConceptStep(?step) \wedge hasSetExpression(?step, ?exp) \wedge$
   $hasFunction(?exp, ?fu) \wedge equals(?fu.type, 'sum') \wedge$
   $subConceptOf(?suburi, ?step.uri, O_{n+1}) \wedge addChildC(?tid, ?newc, ?suburi)$
   $\wedge!(splitC(?stid, ?source, ?suburi) \wedge ChangeAnnotation(?stid, ?tid)$
   $\wedge\, subConceptOf(?source, ?step.uri, O_n) \wedge$
   $!(splitC(?stid, ?source, ?otherSplitUri) \wedge notequals(?otherSplitUri, ?newc)$
   $\wedge\, addChildC(?tid, ?otherSplitUri, ?otherAddUri)$
   $subClassOf(?otherAddUri, ?step.uri, O_{n+1})))$
   $\Rightarrow\; Invalid(?step, 'SubclassAdded')$

2. A concept-step with a value-function gets invalid if one of the subconcepts is renamed or deleted. We will show the rule for the renames here.

   $ConceptStep(?step) \wedge hasSetExpression(?step, ?exp) \wedge$
   $hasFunction(?exp, ?fu) \wedge equals(?fu.type, 'value') \wedge$
   $subConceptOf(?oldUri, ?step.uri, O_{n+1}) \wedge$
   $renameConcept(_, ?oldUri, ?newUri)$
   $\Rightarrow\; Invalid(?step, 'Value - Changed')$

7.5 Proof of Concept Implementation

We have implemented the approach for computing semantic invalidations of annotation paths using the Jena API[3] and the pellet[4] reasoner. The input for the algorithm consists of a set of annotation paths, a set of dependency-definition paths, and the change representation in form of ontology instances as described in section 4.5.3. The output is a subset of the input annotation path where the semantics has (possibly) changed. Additionally, explanations for the semantic invalidations are provided. The rules as proposed in section 7.4 are implemented in form of SPARQL queries that operate on instances of the change-LOG and the instances of the meta ontology. The required negation is implemented in form of SPARQL filters.
The special property $subClassOf(c1, c2, ?v)$ (and $subpropertyOf(p1, p2, ?v)$ analogously) is realized in form of two distinct properties $subClassOfOld(c1, c2)$ and $subClassofNew(c1, c2)$ that are added to the instances of the source and target ontology version of the ontology meta-model. Most invalidation rules can directly be represented in SPARQL, while some more complex queries need additional post-processing. The prototype demonstrated the feasibility of our approach and was also used to generate the output for the case-study in section 8.7.

7.6 Related Work

In [37] the consistent evolution of OWL ontologies is addressed. The authors describe structural, logical and user-defined consistency requirements. While the structural and logical requirements are directly defined by the used ontology formalism the user-defined requirements describe additional requirements from the application domain that cannot be expressed with the underlying ontology language. The authors do not make any suggestion on how these requirements should be expressed. Our approach can be seen as one specific form of user-defined-consistency requirements. The main difference is that in our case the artifact that becomes inconsistent if a user-defined consistency-definition is violated is not the ontology itself but instance-data in XML-documents. In [9] functional dependencies over Aboxes (individuals) are addressed. The dependencies are formulated in the form antecedent, consequent and an optional deterministic function. The antecedent and consequent are formulated via path expressions which can be compared to our approach. The dependencies are directly transformed to SWRL-rules. Therefore, the functional dependencies directly operate on the individuals (Abox) and additional knowledge can be added to the Abox. In addition, data that does not comply with the rules can be marked as inconsistent. In contrast to our approach, the approach is limited to the instance layer which makes it unusable for our scenario where instance-data from XML-documents is never added to the Abox. Therefore, knowledge about changes needs to be evaluated on class-level in order to predict semantic-changes of the semantics of instance-data.

[3] http://incubator.apache.org/jena/
[4] http://clarkparsia.com/pellet/

In [83] the validity of data-instances after ontology evolution is evaluated. An algorithm is proposed that takes a number of explicit changes as input and calculates the implicit changes that are induced by the explicit changes. These explicit and implicit changes can then be used to track the validity of data-instances. The general idea of the approach is that if an artifact gets more restricted existing instances are invalidated. Since the approach only takes into account implicit changes that can be computed based on explicit changes it does not support the explicit definition of change-dependencies.

While there is only limited work on dependencies in the field of ontologies it is traditionally broadly studied in the database community. Recent and related research in this field is for example [8] and [14]. In [8] a model and system is presented that keeps track of the provenance of data that is copied from different (possibly curated) databases to some curated database. Changes in the source databases may influence the data in the target databases. Therefore, provenance information is required to track those changes. In [14] the problem of provenance in databases is formalized with an approach that is inspired by dependency analysis techniques known from program analysis or slicing [43] techniques. In contrast to our approach both provenance approaches cope with changes of instance data and do not address changes of schema/meta-data.

7.7 Conclusion

In this chapter we have addressed the problem of semantic changes of annotations that occur due to the evolution of their reference ontologies. Those changes result in the miss-interpretation of instance data. It is therefore, necessary to detect such changes. We have first shown that extra knowledge is required to detect such invalidations. We have proposed to model this knowledge explicitly by using change-dependency definitions. Such definitions state that an annotations/ontology element is dependent on changes over other artifacts in the ontology. When specific changes occur on those artifacts, the dependent annotation is considered to be (possibly) semantically invalid. We have proposed the definition of change-dependencies and have shown how invalidations can be tracked by evaluating the dependency-definitions and a representation of the changes using standard rules.

Chapter 8

Case Study

In this chapter we provide an imaginary case from the field of the automotive industry that intends to show how the methods and implementations for semantic annotation, mapping generation, change representation, structural and logical maintenance of annotations as well as the detection of semantic changes can be combined. The aforementioned building-blocks are the basis for the future implementation of a tool-set and are shown graphically in figure 8.1.

8.1 The Setting

A number of car-makers and their suppliers have agreed on a common reference ontology. The ontology has two main purposes: On the one hand it is used to allow interoperability between the companies and on the other hand it also represents standards for the classification of cars, options for cars and their evolution. We narrow our case to the exchange of documents that describe a limited set of features and specifications of the offered cars. Those documents are used as a data-source for product descriptions for electronic catalogs as well as printed advertising folders. The structure of the documents is defined by XML-Schemas. The involved partners are car-makers and advertising agencies. Each partner annotates their schemas in order to allow the mapping generation as proposed in chapter 3. Those annotations need to be maintained, when structural or logical errors occur. In addition, changes of the classification of the options and cars may have semantic consequences for the instance data which must be detected in order to prevent legal problems and to classify options and cars according to the current standard.

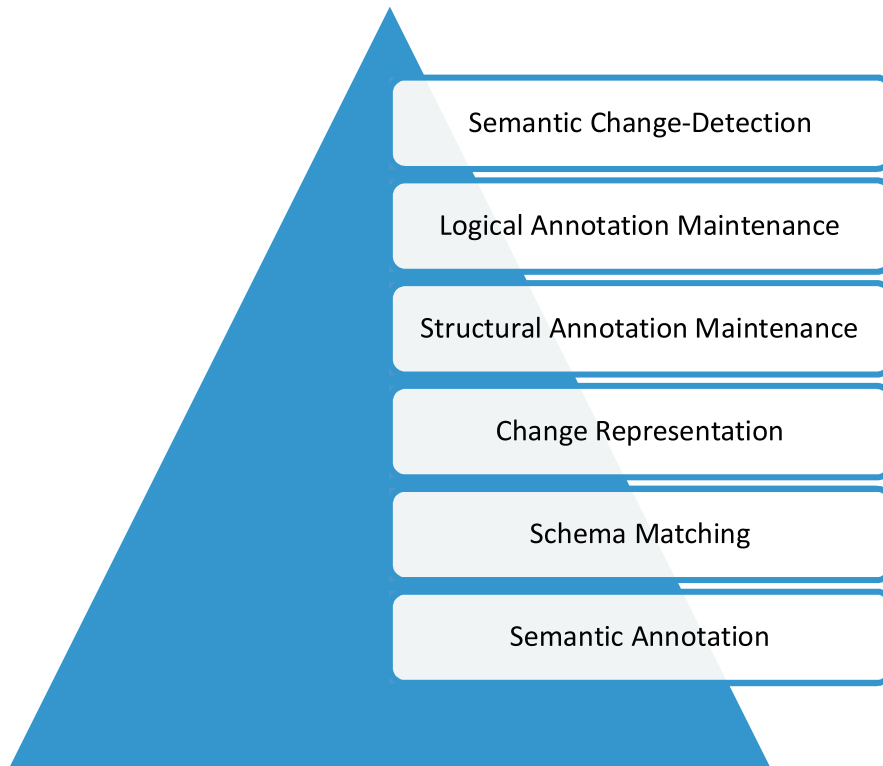

Figure 8.1: Building-blocks for semantic annotations in an evolving environment

8.2 Example Ontology

We propose that the different partners have agreed on a common reference ontology. The relevant parts of the ontology are shown in figure 8.2. We use UML for a compact graphical representation. We use UML classes for concepts, UML attributes for datatype properties and binary relations for object properties. Instances are assigned to concepts via dashed lines.

In addition to the plain OWL reference ontology the partners have agreed on the following change-dependencies that are used to track the consequences of changes in the reference ontology:

- *ComfortOption depends on instances(ComfortOption)*
- *SafetyOption depends on instances(SafetyOption)*
- *CarMaker/FleetCarEmmissions depends on Sum(Subclasses(PassengerCar))/Co2perKM*

The idea is that the reference ontology is used to classify available options. These options are represented as instance of the reference ontology and changes over those instances have

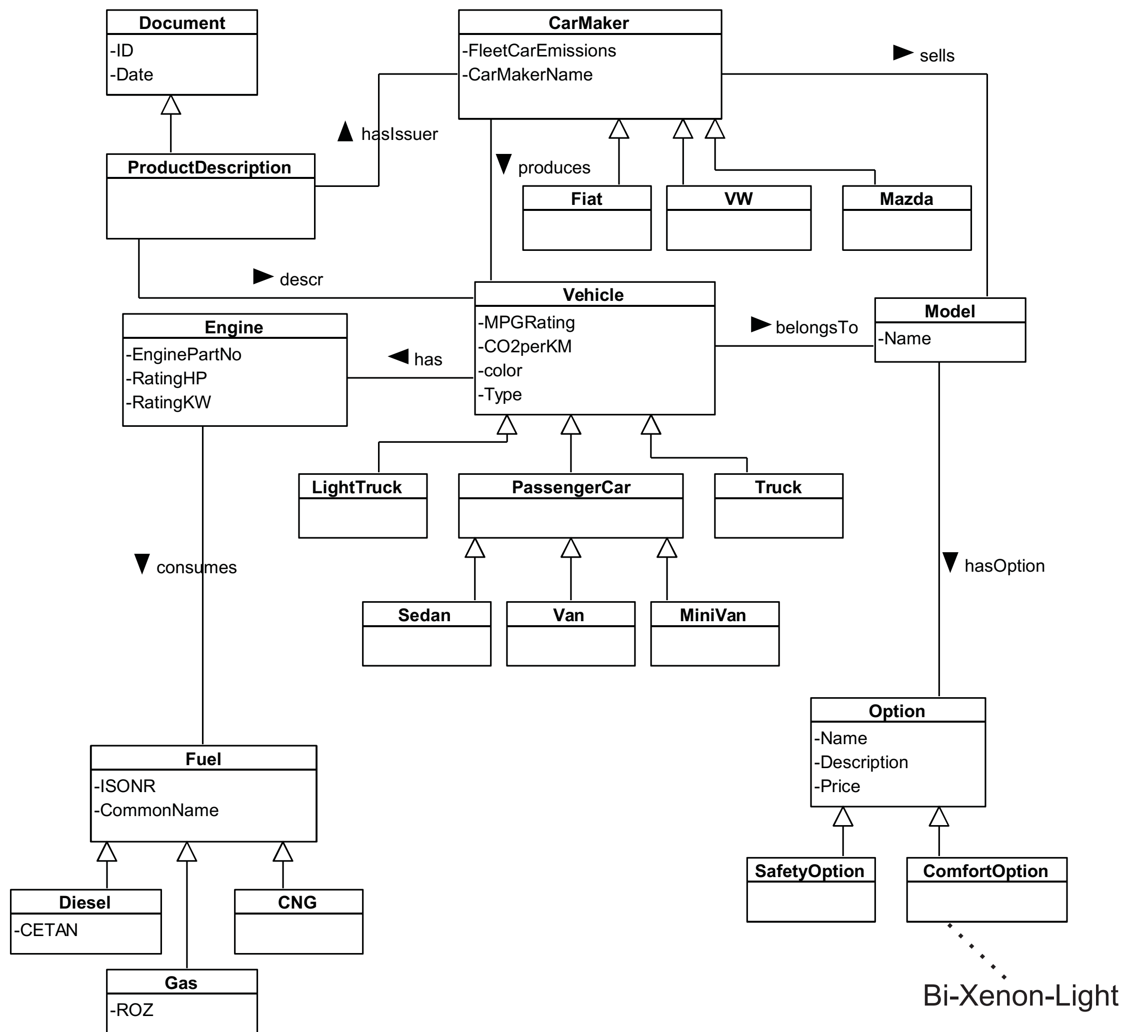

Figure 8.2: Example car ontology first version

consequences for the documents. Therefore, the users should be warned, if they have schemas that potentially contain options that were subject to a change/reclassification. In addition, a general change-dependency on the average CO_2-Emissions of the car-makers is defined according to some law. It states, that the emissions of a car-maker depend on the sum of the emissions of the subclasses of passenger-cars. Since no passenger-car is actually added this is an abstract sum, that cannot return any value. But changes of the subclasses have consequences on the values in the documents. Thus, also in this case the user should be warned about modification that have influence on the average CO_2 emissions.

8.3 Example Annotations

We will concentrate on a small use-case between two partners, where one specific car-maker sends documents with the specifications and available options to an advertising agency. The schema that is used by the car-maker is annotated with the annotation set in listing 8.1. The set of annotations of the schema of the advertising agency is shown in listing 8.2. The car-maker *Mazda* has annotated the elements specifically for the semantics for documents from *Mazda*. For example *Mazda* uses the schema only for passenger cars and the issuer is always the car-maker itself. In contrast the advertising agency uses a broader annotation that allows all kinds of vehicles from arbitrary car-makers.

```
1  ProductDescription/ID
2  ProductDescription/Date
3  ProductDescription/hasIssuer/Mazda/FleetCarEmissions
4  ProductDescription/hasIssuer/Mazda/CarmakerName
5  ProductDescription/descr/PassengerCar/belongsTo/Model/Name
6  ProductDescription/descr/PassengerCar/Type
7  ProductDescription/descr/PassengerCar/has/Engine/RatingHP
8  ProductDescription/descr/PassengerCar/has/Egnine/consumes/Fuel/ISONR
9  ProductDescription/descr/PassengerCar/has/Egnine/consumes/Fuel/CommonName
10 ProductDescription/descr/PassengerCar/MPGRating
11 ProductDescription/descr/PassengerCar/color
12 ProductDescription/descr/PassengerCar/belongsTo/Model/hasOption/SafetyOption
13 ProductDescription/descr/PassengerCar/belongsTo/Model/hasOption/ComfortOption
```

Listing 8.1: Example annotations of schema 1

```
1  ProductDescription/ID
2  ProductDescription/Date
3  ProductDescription/hasIssuer/CarMaker/FleetCarEmissions
4  ProductDescription/hasIssuer/CarMaker/CarmakerName
5  ProductDescription/descr/Vehicle/belongsTo/Model/Name
6  ProductDescription/descr/Vehicle/Type
7  ProductDescription/descr/Vehicle/has/Engine/RatingKW
8  ProductDescription/descr/Vehicle/has/Egnine/consumes/Fuel/ISONR
9  ProductDescription/descr/Vehicle/has/Egnine/consumes/Fuel/CommonName
10 ProductDescription/descr/Vehicle/MPGRating
11 ProductDescription/descr/Vehicle/color
12 ProductDescription/descr/PassengerCar/belongsTo/Model/hasOption/SafetyOption
13 ProductDescription/descr/PassengerCar/belongsTo/Model/hasOption/ComfortOption
```

Listing 8.2: Example annotations of schema 2

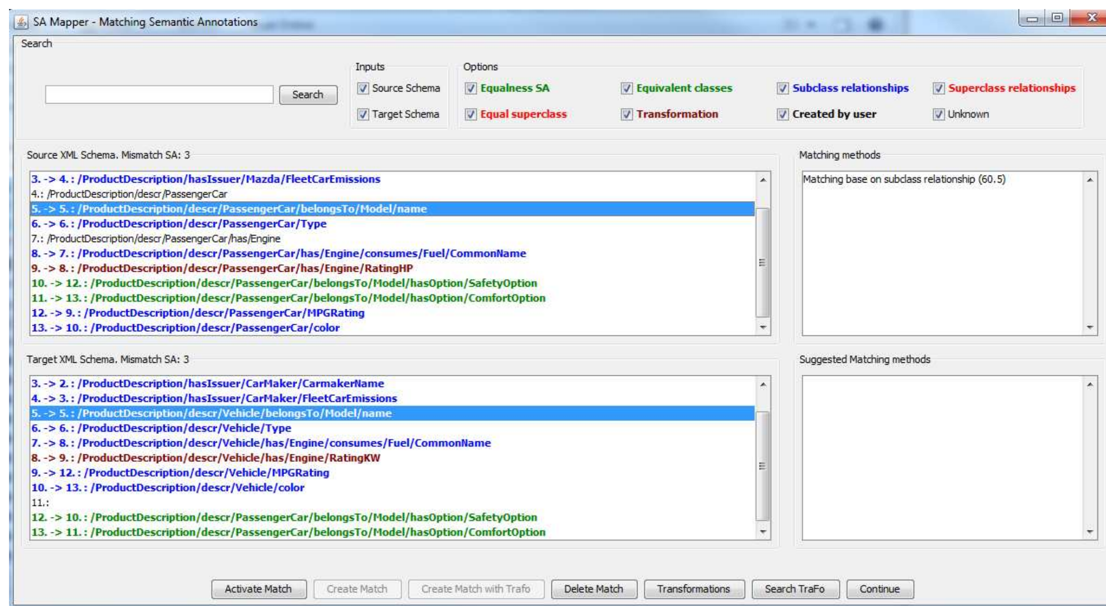

Figure 8.3: Screen-shot of the mapping-prototype

8.4 Mapping Generation

Now the advertising agency needs to generate a mapping between the schema of the car-maker and their own schema. This can be realized fully automatically with the proposed schema mapping methods from chapter 3. Most of the annotations can be matched by subclass or equivalent class matches, only the annotation *...has/Engine/RatingHP* needs to be matched with a complex matching expression that uses a transformation template to translate from horse-power to kilowatts for the target annotation *...has/Engine/RatingKW*. A screen-shot of the prototype implementation is shown in figure 8.3. Most matches in the screen-shot are subclass matches because the source schema is a schema of a specific car-maker and the target schema is a general schema for any type of descriptions of vehicles. The automatically generated mapping (opened in MapForce) is shown in figure 8.4.

8.5 Ontology Changes

At some point in time the common reference ontology is modified to reflect new requirements. The new ontology version is shown in figure 8.5. The following modifications were made: The concept *ProductDescription* is removed. The subclasses *Van* and *Mini-Van* of *passenger-car* are merged to one common concept *Van*. Light-Trucks that were previously considered to be general vehicles are now classified as *PassengerCars*. The reason for that is a change

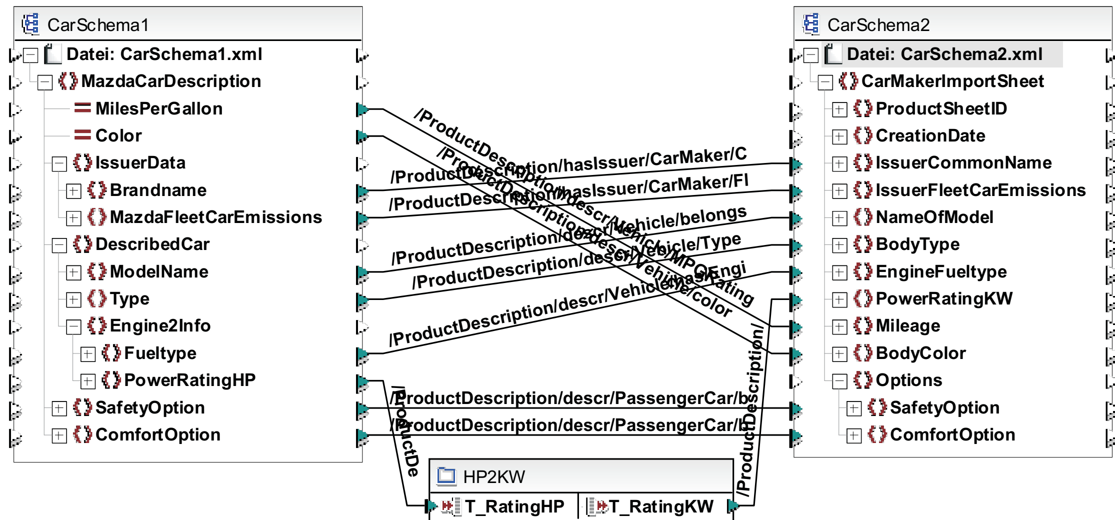

Figure 8.4: Automatically generated mapping

in the legislation. The domain of *hasOption* is changed from *Model* to *Vehicle*. Finally the datatype-property *CarMakerName* is renamed to *BrandName*. This leads to a set of instances of the change ontology that is depicted in figure 8.6. The change ontology instances were generated automatically by comparing both ontology versions using the change representation implementation from chapter 4. There is one composite-change. It has the change-id c_{11} and it defines that *MiniVan* and *Van* are now both considered as *Vans*. There are two atomic changes that are part of this composite change, namely c_{10} that deletes the concept *MiniVan* and c_3 that removes *MiniVan* from *PassengerCar*. Both atomic changes are associated with the composite-change by using the object-property assertion *partOfComplexChange* to c_{11}.

8.6 Structural and Logical Annotation Maintenance

After the ontology has evolved the annotations need to be revalidated structurally, logically and semantically. The structural revalidation (see chapter 5) comes to the conclusion that all annotations are structurally invalid because the concept *ProductDescription* has been removed. Using the proposed algorithms from section 5.2 to find possible repairs, there exists one single exchange-candidate *Document* for *ProductDescription*. Thus, *Document* is now used for all annotations. The next structurally invalid annotation is *Document/hasIssuer/CarMaker/CarmakerName*. According to the change representation *CarmakerName* was renamed to *Brandname*. Thus, the path can simply be replaced with a new path *Document/hasIssuer/CarMaker/Brandname*. Now all annotations are structurally valid and we can check for logical invalidations. The following annotations are not satisfiable, when added to the reference ontology:

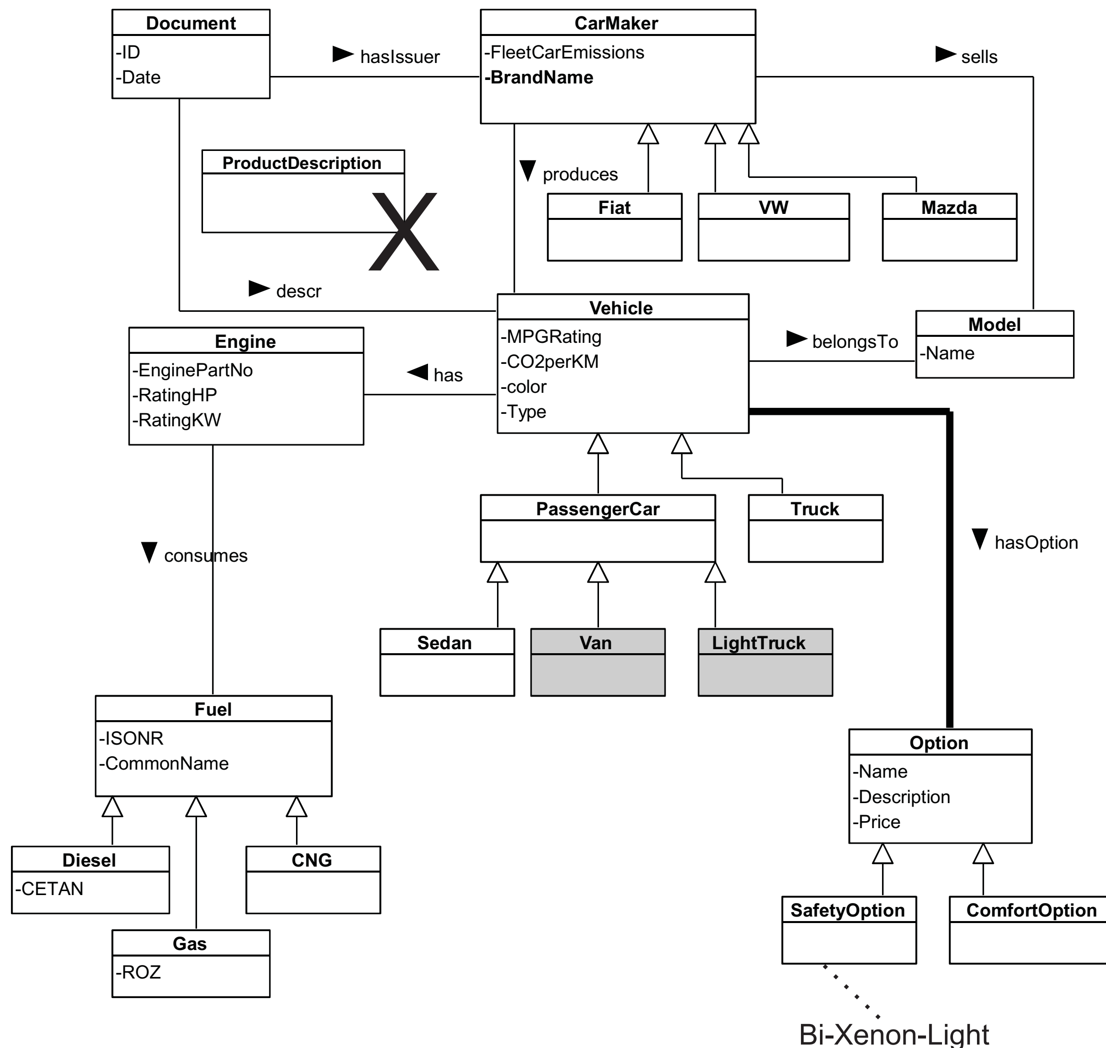

Figure 8.5: Example car ontology second version

- Document/descr/Vehicle/belongsTo/Model/hasOption/SafetyOption
- Document/descr/Vehicle/belongsTo/Model/hasOption/ComfortOption
- Document/descr/PassengerCar/belongsTo/Model/hasOption/SafetyOption
- Document/descr/PassengerCar/belongsTo/Model/hasOption/ComfortOption

Using the methods from chapter 6 we know that the triples *Document/descr/Vehicle*, *Vehicle/belongsTo/Model*, *Document/descr/PassengerCar*, and *PassengerCar/belongsTo/Model* are globally valid triples because they are part of valid annotation paths. As a consequence the triples that are potentially invalid are *Model/hasOption/SafetyOption* and *Model/hasOption/ComfortOption*. A

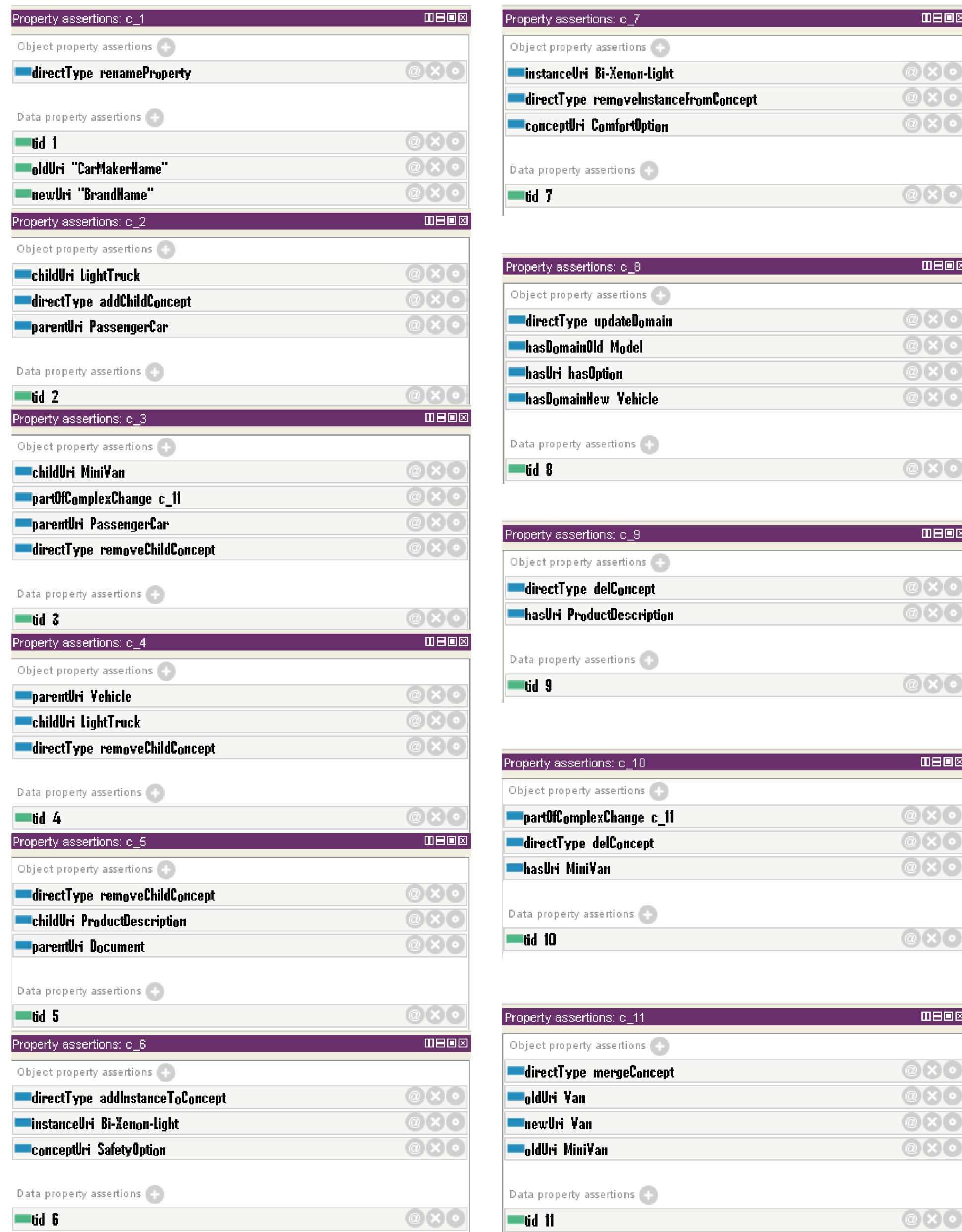

Figure 8.6: Instances of the change ontology

check for local invalidations shows that the domain of *hasOption* is disjoint from *Model*. A query to the change representations uncovers that the domain was changed from *Model* to *Vehicle*. The user can thus, directly change the annotations to the new set of annotations:

- Document/descr/Vehicle/hasOption/SafetyOption

- Document/descr/Vehicle/hasOption/ComfortOption
- Document/descr/PassengerCar/hasOption/SafetyOption
- Document/descr/PassengerCar/hasOption/ComfortOption

8.7 Detection of Semantic Changes

After the annotations are maintained structurally and logically we can check for semantic changes that may have influence on the instance data of the documents. We first detect the relevant change-dependencies for the annotations as defined in section 7.4.

It turns out that *Document/hasIssuer/CarMaker/FleetCarEmmissions* is a subconcept of *CarMaker/FleetCarEmmissions*, which depends on the (virtual) sum of the *hasCo2perKM* of *passengerCars*. According to the rules for the invalidation of steps from subsection 7.4 we get the invalidation report for *Document/hasIssuer/CarMaker/FleetCarEmmissions*: Subclass *Light-Truck* added. The user can now acquire the official new CO_2 emission value for the car-company and replace all values in the instance documents with the new value. The merge of *Mini-Van* and *Van* is not relevant for the dependency definition because is does not change the number of potential instances. Therefore, it is not detected as an invalidation.

The next dependency definitions that are relevant are *ComfortOption* depends on *instances(ComfortOption)* and *SafetyOption* depends on *instances(SafetyOption)*. According to the change-LOG the following report is generated and added to the corresponding annotations.

- *ComfortOption is invalid because the instance Bi-Xenon-Light was removed.*
- *SecurityOption is invalid because the instance Bi-Xenon-Light was added.*

The user can now check, if the documents potentially contain *Bi-Xenon-Lights*. If they potentially contain them, he/she might decide to move them to the corresponding schema elements. The relevant output that was generated by our prototype implementation from section 7.5 is shown in listing 8.3. The denoted rules that triggered the invalidation report refer to the rules in section 7.4.

```
1  Document/descr/Vehicle/hasOption/SafetyOption
2      Checking for Set−Expression
3      Relevant change: addInstanceToConcept SafetyOption Bi−Xenon−Light
4      Step instances(SafetyOption) invalid due to rule 7.
5      Dependency Definition violated: instances(SafetyOption)
6  Document/descr/Vehicle/hasOption/ComfortOption
7      Checking for Set−Expression
8      Relevant change: removeInstanceFromConcept ComfortOption Bi−Xenon−Light
9      Step instances(ComfortOption) invalid due to rule 7.
10     Dependency Definition violated: instances(ComfortOption)
11 CarMaker/FleetCarEmissions
12     Checking for function and set expression
13     Relevant change: addChildConcept LightTruck
14     Step Sum(Subclasses(PassengerCar)) invalid due to rule A1.
```

Listing 8.3: Generated output for the detection of semantic changes

8.8 Conclusion

In this chapter, we have presented an use-case that shows how the methods from this thesis can be used together. We have shown the annotations of the schemas and the automatic mapping generation with our prototype implementation of chapter 3. We have then assumed that the reference ontology was changed. The changes were represented using our change representation and detection approach from chapter 4. The changes made the annotations structurally and logically invalid. We could propose possible repair actions by using the methods for structural and logical maintenance from chapter 5 and 6. Finally we could detect semantic changes with the methods presented in chapter 7.

Chapter 9

Conclusion

In chapter 1 we have motivated and introduced the problem of ontology evolution in a semantic-annotation-based scenario. Semantic annotations can be used to allow interoperability between heterogeneous systems by explicitly defining the semantics of the data with regard to some reference ontology. The reference ontology formally describes the semantics of the specific business domain. This allows the correct interpretation of the semantics of the data by the participating partners/systems. Since the real world constantly evolves it is natural that also the reference ontology needs to adopt to new requirements over time. This leads to the evolution of the reference ontology. The evolution has two consequences: First of all, the annotations need to be maintained when they do not still comply with the new version of the reference ontology. Second, the evolving ontology does not only express the semantics of the domain at some specific point in time. It can also represent the changes of the domain. This knowledge can be used to communicate such changes to the depending partners in order to adopt their systems and data. In this thesis the primary goal was the maintenance of the semantic annotations in the dimensions structure and logics and the detection of semantic changes that have influence on the interpretation of instance data.
In this research we concentrated on the purely declarative annotation of XML-Schemas. Such annotations have two main advantages over lifting and lowering mappings. First, they allow the creation of schema mappings, which provides a good scalability as shown in the evaluation in section 3.5 and second, they allow the detection and (partly) automatic repair of invalidations due to ontology evolution. We have provided the following contributions in this thesis:

- A declarative semantic annotation method for XML-Schemas [57].
- Schema mapping methods based on the proposed annotation method [96].
- Methods for the proper representation of ontology changes [58].

- Methods for the structural and logical maintenance of annotations when the ontology evolves [59].
- Detection methods for semantic changes after ontology evolution [58].

The goal of this thesis was pure research, that should be a basis for the future implementation of a tool-set. Therefore, we we have only partly implemented the proposed approaches in form of proof of concept implementations. In particular we have implemented a schema mapping approach, the detection approach for semantic changes and the change-representation method.

We have introduced a method for the purely declarative semantic annotation of XML-Schemas in chapter 2. The method is based on path expressions that address concepts, properties and instances of the reference ontology. Those path expressions can be used to annotate schema elements according to the SAWSDL proposal. The path expressions can automatically be converted to OWL formulas in order to represent the semantics with the help of the reference ontology.

The goal of the annotations in this research was the automatic generation of transformation scripts that transform instance documents of one schema to instance documents that comply with another schema. This can be realized by schema matching and mapping methods. Therefore, we have introduced the problem of schema matching in chapter 3. Schema matching is used to find correspondences between schemas in order to generate mappings. We have shown how the proposed semantic annotations can be used to find matches. Such matches can be simple matches where one element of the source schema directly corresponds to one element of the target schema or they can be complex matches, where some function relates sets of schema elements of the source schema to sets of elements of the target schema. In order to detect complex matches additional knowledge is required that we represent in form of annotated transformation templates.
In order to show the applicability of the approach we have shown how different types of mismatches between schemas can be resolved using our proposed annotation and matching methods. Finally, we have discussed a prototype implementation and have evaluated the performance of our declarative annotation based schema mapping approach against a lifting/lowering approach that uses standard Semantic Web technologies and frameworks. The results clearly show that our assumption that a declarative annotation based approach provides a better scalability than a lifting/lowering approach was correct. Our implementation outperformed the lifting/lowering implementation with regard to transformation time per document by a factor of up to 1:130.

An evolutionary reference ontology requires the representation of the changes between the different ontology versions. In chapter 4, we have first discussed the requirements for change representation for this thesis and have then provided a survey on approaches in this

field. We came to the conclusion that none of the approaches could fulfill all requirements and we have therefore, proposed a hybrid, meta ontology based approach that provides atomic and composite declarative change descriptions over the ontology graph in form of change ontology instance. These change instances directly relate instances of a meta ontology of the old and the new ontology version. This approach allows reasoning support over the changes and both ontology versions by using standard reasoners and rule languages. We have also presented the implementation of this approach. This high-level change representation is accomplished with a complete axiom-based change-log which can be used for the justification of logical invalidations.

When the ontology evolves also the annotations need to be adopted. We have first proposed a complete language to modify the annotation paths in chapter 5. As a next step we have discussed the problem of structural invalidation of annotation paths and provided algorithms to check the paths for validity and to detect the type of invalidation. An invalid path consists of at least one invalid step. An invalid step can be repaired by replacing the referenced element with an existing element of the new ontology version. Such a replacement can be computed by evaluating the changes between the old an the new ontology version and may require to replace a more specific element of the old ontology version with a less specific element of the new ontology version. We assume that a good replacement candidate is a replacement where a minimum number of such abstractions is required. When the possible repairs for each invalid step of an invalid path are computed those solutions can be used to propose replacement candidates for the complete annotation path. The best replacement candidates are those that require a minimum overall number of abstractions. We have finally introduced an additional annotation maintenance scenario in a collaborative environment that limits the human intervention by using mapping composition.

In addition to structural invalidations logical invalidations can occur. A logical invalid annotation path expression is a structurally valid path expression, where the corresponding annotation concept is unsatisfiable in the reference ontology. We have provided an in depth analysis of the possible causes for logically invalid annotations in chapter 6. There are basically two types of logical invalidations: local invalidations and non-local invalidations. We have provided efficient detection methods for local and non-local invalidations. Those techniques can be used for new annotations as well as for the maintenance of annotations after ontology evolution. Our method not only identifies the annotations which are logically invalid, but also narrows the inspection area to the shortest possible path and gives indications on the causes of the invalidation for annotation creation and maintenance. The proposed algorithms are built upon the functionality usually provided by generic reasoners for OWL ontologies, so they are not restricted to a specific reasoner or ontology management system.

A logically valid concept does not yet guarantee that the semantics of the annotated elements did not change. We have introduced the problem in chapter 7. We have shown that the plain OWL ontology is not sufficient to detect such semantic changes. Therefore, we have proposed to add explicit change-dependencies to the ontology. Those change-dependencies define, when a concept or annotation gets invalid with regard to changes over other ontology elements. We have proposed rules for the detection of such invalidations and presented an implementation that directly operates on the change representation of chapter 4.

Finally we have shown how all the contributions for annotation, schema mapping, change representation, annotation maintenance and semantic-change detection can be combined in an illustrative case study in chapter 8.

Chapter 10

Appendix

```
1  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:sawsdl="http://www.w3.org/ns/sawsdl" elementFormDefault="qualified" attributeFormDefault="unqualified">
3    <xs:element name="invoice" sawsdl:modelReference="/InvoiceDocument">
4      <!--
5    This is an XML-Schema that is used by a company that sells only furniture. In addition this company sells its good only to private customers.
6    -->
7      <xs:complexType>
8        <xs:sequence>
9          <xs:element name="BuyerInfo" sawsdl:modelReference="/InvoiceDocument/hasBuyer/PrivateBuyer">
10           <xs:complexType>
11             <xs:sequence>
12               <xs:element name="Firstname" sawsdl:modelReference="/InvoiceDocument/hasBuyer/PrivateBuyer/hasFirstname/"/>
13               <xs:element name="LastName" sawsdl:modelReference="/InvoiceDocument/hasBuyer/PrivateBuyer/hasLastName"/>
14               <xs:element name="Address" type="AddressType" sawsdl:modelReference="/InvoiceDocument/hasBuyer/PrivateBuyer/hasContact/PostalAddress"/>
15             </xs:sequence>
16           </xs:complexType>
17         </xs:element>
18         <xs:element name="SelllerInfo" sawsdl:modelReference="/InvoiceDocument/hasSeller/Seller">
19           <xs:complexType>
20             <xs:sequence>
21               <xs:element name="SellerCompany" sawsdl:modelReference="/InvoiceDocument/hasSeller/Seller/hasCompany/Company"/>
22               <xs:element name="Address" type="AddressType" sawsdl:modelReference="/InvoiceDocument/hasSeller/Seller/hasContact/PostalAddress"/>
23             </xs:sequence>
24           </xs:complexType>
25         </xs:element>
26         <xs:element name="Products">
27           <xs:complexType>
28             <xs:choice>
29               <xs:element name="Product" maxOccurs="unbounded" sawsdl:modelReference="/InvoiceDocument/hasProductList/ProductList/hasItem/ListItem">
30                 <xs:complexType>
31                   <xs:sequence>
32                     <xs:element name="FurnitureID" sawsdl:modelReference="/InvoiceDocument/hasProductList/ProductList/hasItem/ListItem/hasProduct/Furniture/hasID"/>
33                     <xs:element name="FurnitureDescription" sawsdl:modelReference="
   /InvoiceDocument/hasProductList/ProductList/hasItem/ListItem/hasProduct/Furniture/hasDescription"/>
34                     <xs:element name="quantity" sawsdl:modelReference="/InvoiceDocument/hasProductList/ProductList/hasItem/ListItem/hasQuantity"/>
35                     <xs:element name="singleNettopriceEuro" sawsdl:modelReference="
   /InvoiceDocument/hasProductList/ProductList/hasItem/ListItem/hasPrice/NettoItemPrice/hasEuroValue"/>
36                   </xs:sequence>
37                 </xs:complexType>
38               </xs:element>
39             </xs:choice>
40           </xs:complexType>
```

Figure 1: Source of mismatch example source schema page 1

```
41              </xs:element>
42              <xs:element name="GuaranteeInfo">
43               <xs:complexType>
44                <xs:attribute name="Duration" sawsdl:modelReference="/InvoiceDocument/declares/GenericGuarantee/hasDuration"/>
45                <xs:attribute name="Type" sawsdl:modelReference="/InvoiceDocument/declares/GenericGuarantee/hasType"/>
46              </xs:complexType>
47              </xs:element>
48                <xs:element name="invoiceMetaData">
49                <xs:complexType>
50                  <xs:attribute name="InvoiceNumber" sawsdl:modelReference="/InvoiceDocument/hasID"/>
51                  <xs:attribute name="InvoiceDate" sawsdl:modelReference="/InvoiceDocument/hasDate"/>
52                </xs:complexType>
53              </xs:element>
54            </xs:sequence>
55          </xs:complexType>
56        </xs:element>
57        <xs:complexType name="AddressType">
58          <xs:sequence>
59            <xs:element name="Street" sawsdl:modelReference="/PostalAddress/hasStreet"/>
60            <xs:element name="zip" sawsdl:modelReference="/PostalAddress/hasZip"/>
61            <xs:element name="City" sawsdl:modelReference="/PostalAddress/hasCity"/>
62            <xs:element name="Country" sawsdl:modelReference="/PostalAddress/hasCountry"/>
63          </xs:sequence>
64        </xs:complexType>
65      </xs:schema>
66
```

Figure 2: Source of mismatch example source schema page 2

```
1  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:sawsdl="http://www.w3.org/ns/sawsdl" elementFormDefault="qualified" attributeFormDefault="unqualified">
3    <!--
4    This is an XML-Schema that is used by a company that sells any kind of good. In addition this company sells its good to any kind of customer.
5    -->
6    <xs:element name="invoice" sawsdl:modelReference="/InvoiceDocument">
7      <xs:complexType>
8        <xs:sequence>
9          <xs:element name="InvolvedParties" sawsdl:modelReference="/InvoiceDocument/hasInvolvedParty/BusinessParty/">
10           <xs:complexType>
11             <xs:sequence>
12               <xs:element name="BuyerFirstAndLastName" sawsdl:modelReference="/InvoiceDocument/hasBuyer/Buyer/hasFullName/"/>
13               <xs:element name="SellerCompanyName" sawsdl:modelReference="/InvoiceDocument/hasSeller/Seller/hasCompany/Company"/>
14               <xs:element name="StreetOfBuyer" sawsdl:modelReference="/InvoiceDocument/hasBuyer/Buyer/hasContact/PostalAddress/hasStreet"/>
15               <xs:element name="zipOfBuyer" sawsdl:modelReference="/InvoiceDocument/hasBuyer/Buyer/hasContact/PostalAddress/hasZip"/>
16               <xs:element name="CityOfBuyer" sawsdl:modelReference="/InvoiceDocument/hasBuyer/Buyer/hasContact/PostalAddress/hasCity"/>
17               <xs:element name="CountryOfBuyer" sawsdl:modelReference="/InvoiceDocument/hasBuyer/Buyer/hasContact/PostalAddress/hasCountry"/>
18               <xs:element name="StreetOfSeller" sawsdl:modelReference="/InvoiceDocument/hasSeller/Seller/hasContact/PostalAddress/hasStreet"/>
19               <xs:element name="zipOfSeller" sawsdl:modelReference="/InvoiceDocument/hasSeller/Seller/hasContact/PostalAddress/hasZip"/>
20               <xs:element name="CityOfSeller" sawsdl:modelReference="/InvoiceDocument/hasSeller/Seller/hasContact/PostalAddress/hasCity"/>
21               <xs:element name="CountryOfSeller" sawsdl:modelReference="/InvoiceDocument/hasSeller/Seller/hasContact/PostalAddress/hasCountry"/>
22             </xs:sequence>
23           </xs:complexType>
24         </xs:element>
25         <xs:element name="InvoiceItem" maxOccurs="unbounded" sawsdl:modelReference="/InvoiceDocument/hasProductList/ProductList/hasItem/ListItem">
26           <xs:complexType>
27             <xs:attribute name="ProductId" sawsdl:modelReference="/InvoiceDocument/hasProductList/ProductList/hasItem/ListItem/hasProduct/Product/hasID"/>
28             <xs:attribute name="ProductDescription" sawsdl:modelReference="/InvoiceDocument/hasProductList/ProductList/hasItem/ListItem/hasProduct/Product/hasDescription
   "/>
29             <xs:attribute name="quantity" sawsdl:modelReference="/InvoiceDocument/hasProductList/ProductList/hasItem/ListItem/hasQuantity"/>
30             <xs:attribute name="singleNettopriceUSD" sawsdl:modelReference="
   /InvoiceDocument/hasProductList/ProductList/hasItem/ListItem/hasPrice/NettoItemPrice/hasDollarValue"/>
31           </xs:complexType>
32         </xs:element>
33         <xs:choice>
34             <xs:element name="GoldGuarenteeDuration" sawsdl:modelReference="/InvoiceDocument/declares/GoldGuarentee/hasDuration"></xs:element>
35             <xs:element name="PlatinGuarenteeDuration" sawsdl:modelReference="/InvoiceDocument/declares/PlatinGuarentee/hasDuration"></xs:element>
36         </xs:choice>
37       </xs:sequence>
38       <xs:attribute name="InvoiceNumber" sawsdl:modelReference="/InvoiceDocument/hasID"/>
39       <xs:attribute name="InvoiceDate" sawsdl:modelReference="/InvoiceDocument/hasDate"/>
40     </xs:complexType>
41   </xs:element>
42 </xs:schema>
43
```

Figure 3: Source of mismatch example target schema

Bibliography

[1] Alsayed Algergawy, Richi Nayak, and Gunter Saake. Xml schema element similarity measures: A schema matching context. In *Proceedings of the Confederated International Conferences, CoopIS, DOA, IS, and ODBASE 2009 on On the Move to Meaningful Internet Systems: Part II*, OTM '09, pages 1246–1253, Berlin, Heidelberg, 2009. Springer-Verlag. 27

[2] David Aumueller, Hong-Hai Do, Sabine Massmann, and Erhard Rahm. Schema and ontology matching with COMA++. In *Proc. of the 2005 ACM SIGMOD Int'l. Conf. on Management of Data*, pages 906–908, 2005. 21, 23, 24, 30, 94

[3] Dave Beckett and Jeen Broekstra. SPARQL query results XML format. W3C recommendation, W3C, January 2008. http://www.w3.org/TR/2008/REC-rdf-sparql-XMLres-20080115/. 49

[4] Domenico Beneventano, Sabina El Haoum, and Daniele Montanari. Mapping of heterogeneous schemata, business structures, and terminologies. In *DEXA '07: Proceedings of the 18th International Conference on Database and Expert Systems Applications*, pages 412–418, Washington, DC, USA, 2007. IEEE Computer Society. 19

[5] David Booth and Canyang Kevin Liu. Web services description language (WSDL) version 2.0 part 0: Primer. W3C recommendation, W3C, June 2007. http://www.w3.org/TR/2007/REC-wsdl20-primer-20070626. 3

[6] Tim Bray, Jean Paoli, Eve Maler, François Yergeau, and C. M. Sperberg-McQueen. Extensible markup language (XML) 1.0 (fifth edition). W3C recommendation, W3C, November 2008. http://www.w3.org/TR/2008/REC-xml-20081126/. 7

[7] Dan Brickley and Ramanathan V. Guha. Rdf vocabulary description language 1.0: Rdf schema. W3C recommendation, W3C, 2 2004. 9, 19, 62

[8] Peter Buneman, Adriane P. Chapman, and James Cheney. Provenance management in curated databases. In *Proc. of SIGMOD'06*, pages 539–550. ACM, 2006. 131

[9] Jean-Paul Calbimonte, Fabio Porto, and C. Maria Keet. Functional dependencies in owl abox. In Angelo Brayner, editor, *Proc. of SBBD'09*, pages 16–30. SBC, 2009. 130

[10] Giorgio Callegari, Michele Missikoff, Osimi M, and Francesco Taglino. Semantic annotation language and tool for information and business processes - appendix f: User manual, athena deliverable d.a3.3 available at the leks (laboratory for enterprise knowledge and systems) web site http://leks-pub.iasi.cnr.it/astar/. Technical report. 19

[11] Silvana Castano, Valeria De Antonellis, and Sabrina De Capitani di Vimercati. Global viewing of heterogeneous data sources. *IEEE Trans. on Knowl. and Data Eng.*, 13:277–297, March 2001. 21

[12] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. *SIGMOD Rec.*, 25:493–504, June 1996. 63

[13] Chuming Chen and Manton M. Matthews. A new approach to managing the evolution of owl ontologies. In Hamid R. Arabnia and Andy Marsh, editors, *SWWS*, pages 57–63. CSREA Press, 2008. 68, 71, 72, 73

[14] James Cheney, Amal Ahmed, and Umut Acar. Provenance as dependency analysis. In Marcelo Arenas and Michael Schwartzbach, editors, *Database Programming Languages*, volume 4797 of *LNCS*, pages 138–152. Springer, 2007. 131

[15] James Clark. XSL transformations (XSLT) version 1.0. W3C recommendation, W3C, November 1999. http://www.w3.org/TR/1999/REC-xslt-19991116. 4, 48

[16] Oscar Corcho, Catherine Roussey, Luis Manuel Vilches Blazquez, and Ivan Perez. Pattern-based owl ontology debugging guidelines. In *Proceedings of WOP2009 collocated with ISWC2009*, volume 516. CEUR-WS.org, November 2009. 116

[17] Oscar Corcho, Catherine Roussey, Ondrej Zamazal, and francois scharffe. SPARQL-based Detection of Antipatterns in OWL Ontologies, October 2010. Proceedings of the EKAW2010 Poster and Demo Track. 116

[18] Mathieu d'Aquin, Anne Schlicht, Heiner Stuckenschmidt, and Marta Sabou. Ontology modularization for knowledge selection: Experiments and evaluations. In *DEXA*, pages 874–883, 2007. 120

[19] Mike Dean and Guus Schreiber. OWL web ontology language reference. W3C recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-owl-ref-20040210/. 2, 9, 11

[20] Robin Dhamankar, Yoonkyong Lee, AnHai Doan, Alon Halevy, and Pedro Domingos. imap: discovering complex semantic matches between database schemas. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, pages 383–394, New York, NY, USA, 2004. ACM. 25, 26

[21] Hong-Hai Do and Erhard Rahm. Coma: a system for flexible combination of schema matching approaches. In *Proceedings of the 28th international conference on Very Large Data Bases*, VLDB '02, pages 610–621. VLDB Endowment, 2002. xiii, 21, 23, 24, 28, 29, 30

[22] AnHai Doan, Pedro Domingos, and Alon Y. Halevy. Reconciling schemas of disparate data sources: a machine-learning approach. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, SIGMOD '01, pages 509–520, New York, NY, USA, 2001. ACM. 21

[23] Fabien Duchateau, Zohra Bellahsene, and Remi Coletta. A flexible approach for planning schema matching algorithms. In *Proceedings of the OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008. Part I on On the Move to Meaningful Internet Systems:*, OTM '08, pages 249–264, Berlin, Heidelberg, 2008. Springer-Verlag. 21, 23

[24] Johann Eder and Julius Koepke. Towards semantic interoperability in an evolving environment. In *Proceedings of the 15th International Conference on Concurrent Enterprising (ICEt'09)*, 2009. 1, 3

[25] Johann Eder and Christian Koncilia. Modelling changes in ontologies. In *On The Move - Federated Conferences (OTM 2004)*, pages 662–673, Agia Napa, Cyprus, 10 2004. Springer Verlag. 68, 71, 72, 73

[26] Johann Eder, Marek Lehmann, Christian Koncilia, and Horst Pichler. Using ontologies to compose transformations of xml schema based documents. In *16th Conference on Advanced Information Systems Engineering (CAiSE 2004), INTEROP-EMOI Workshop*, pages 315–318, Riga, Latvia, 6 2004. Faculty of Computer Science and Information Technology. 21

[27] Johann Eder and Karl Wiggisser. Change detection in ontologies using dag comparison. In John Krogstie, Andreas Opdahl, and Guttorm Sindre, editors, *Proc. of CAiSE'07*, volume 4495 of *LNCS*, pages 21–35. Springer, 2007. 63, 71, 72, 73, 79

[28] David W. Embley, Li Xu, and Yihong Ding. Automatic direct and indirect schema mapping: experiences and lessons learned. *SIGMOD Rec.*, 33:14–19, December 2004. 26

[29] Jérôme Euzenat. An api for ontology alignment. In Sheila McIlraith, Dimitris Plexousakis, and Frank van Harmelen, editors, *The Semantic Web - ISWC 2004*, volume 3298 of *Lecture Notes in Computer Science*, pages 698–712. Springer Berlin / Heidelberg, 2004. 69, 71, 72

[30] Richard Fikes and Tom Kehler. The role of frame-based representation in reasoning. *Commun. ACM*, 28:904–920, September 1985. 8

[31] Fausto Giunchiglia, Maurizio Marchese, and Ilya Zaihrayeu. Encoding classifications into lightweight ontologies. In *ESWC*, pages 80–94, 2006. 63

[32] Birte Glimm, Sebastian Rudolph, and Johanna Völker. Integrated metamodeling and diagnosis in owl 2. In *Proceedings of the 9th international semantic web conference on The semantic web - Volume Part I*, ISWC'10, pages 257–272, Berlin, Heidelberg, 2010. Springer-Verlag. 57

[33] Herbert Groiss and Johann Eder. Workflow systems for inter-organizational business processes. *SIGGROUP Bull.*, 18:23–26, December 1997. 1

[34] Anika Groß, Michael Hartung, Toralf Kirsten, and Erhard Rahm. Mapping Composition for Matching Large Life Science Ontologies. In *Proceedings of the 2nd International Conference on Biomedical Ontology*, ICBO 2011, 2011. 94

[35] Enrico Del Grosso, Michele Missikoff, Fabrizio Smith, and Francesco Taglino. Semantic services for business documents reconciliation. In *Proceedings of the WORKSHOP Interoperability through Semantic Data and Service Integration Co-located with SEBD*, pages 1–8, Camogli (Genova), Italy, 2009. 2

[36] Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowl. Acquis.*, 5(2):199–220, 1993. 8

[37] Peter Haase and Ljiljana Stojanovic. Consistent evolution of owl ontologies. In Asunción Gómez-Pérez and Jérôme Euzenat, editors, *The Semantic Web: Research and Applications*, volume 3532 of *Lecture Notes in Computer Science*, pages 91–133. Springer Berlin / Heidelberg, 2005. 10.1007/1143105313. 61, 71, 72, 87, 130

[38] Michael Hartung, Anika Gross, and Erhard Rahm. Rule-based generation of diff evolution mappings between ontology versions. *CoRR*, abs/1010.0122, 2010. 64, 71, 72, 73

[39] Bin He and Kevin Chen-Chuan Chang. Automatic complex schema matching across web query interfaces: A correlation mining approach. *ACM Trans. Database Syst.*, 31:346–395, March 2006. 25, 26

[40] Zhisheng Huang and Heiner Stuckenschmidt. Reasoning with multi-version ontologies: A temporal logic approach. In Yolanda Gil, Enrico Motta, V. Richard Benjamins, and Mark A. Musen, editors, *The Semantic Web - ISWC 2005*, number 3729 in LNCS, pages 398–412, Glaway, Ireland, November 2005. Springer. 70, 71, 72, 73

[41] Buhwan Jeong, Daewon Lee, Hyunbo Cho, and Jaewook Lee. A novel method for measuring semantic similarity for xml schema matching. *Expert Syst. Appl.*, 34:1651–1658, April 2008. 27

[42] Haifeng Jiang, Howard Ho, Lucian Popa, and Wook-Shin Han. Mapping-driven xml transformation. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 1063–1072, New York, NY, USA, 2007. ACM. 11, 19

[43] Longfei Jin and Lei Liu. An ontology slicing method based on ontology definition metamodel. In *BIS*, pages 209–219, 2007. 131

[44] G. Friedrich K. Shchekotykhin, P. Rodler. Balancing brave and cautions query strategies in ontology debugging. In *Proceedings of DX-2011 Workshop*, pages 122–130. DX Society, 2011. 116

[45] Aditya Kalyanpur, Bijan Parsia, Matthew Horridge, and Evren Sirin. Finding all justifications of owl dl entailments. In Karl Aberer, Key-Sun Choi, Natasha Fridman Noy, Dean Allemang, Kyung-Il Lee, Lyndon J. B. Nixon, Jennifer Golbeck, Peter Mika, Diana Maynard, Riichiro Mizoguchi, Guus Schreiber, and Philippe Cudré-Mauroux, editors, *ISWC/ASWC*, volume 4825 of *Lecture Notes in Computer Science*, pages 267–280. Springer, 2007. xiii, 113, 114, 116

[46] Asad Masood Khattak, Khalid Latif, Manhyung Han, Sungyoung Lee, Young-Koo Lee, and Hyoung-Il Kim. Change tracer: Tracking changes in web ontologies. In *ICTAI*, pages 449–456. IEEE Computer Society, 2009. 66, 71, 72

[47] Asad Masood Khattak, Khalid Latif, Sharifullah Khan, and Nabeel Ahmed. Managing change history in web ontologies. *Semantics, Knowledge and Grid, International Conference on*, 0:347–350, 2008. 65

[48] Toralf Kirsten, Michael Hartung, Anika Gross, and Erhard Rahm. Efficient management of biomedical ontology versions. In Robert Meersman, Pilar Herrero, and Tharam S. Dillon, editors, *OTM Workshops*, volume 5872 of *Lecture Notes in Computer Science*, pages 574–583. Springer, 2009. 69, 71, 72

[49] Michael Klein and Dieter Fensel. Ontology versioning on the semantic web. In *Proc. 1st Int. Semantic Web Working Symp.*, pages 75–91, Stanford University, CA, USA, 2001. 62

[50] Michael Klein, Dieter Fensel, van Harmelen Frank, and Ian Horrocks. The relation between ontologies and xml schemata, August 2000. 20

[51] Michel C. A. Klein, Dieter Fensel, Atanas Kiryakov, and Damyan Ognyanov. Ontology versioning and change detection on the web. In *EKAW '02: Proceedings of the 13th International Conference on Knowledge Engineering and Knowledge Management. Ontologies and the Semantic Web*, pages 197–212, London, UK, 2002. Springer-Verlag. 62, 72, 73

[52] Peep Küngas and Marlon Dumas. Cost-effective semantic annotation of xml schemas and web service interfaces. *Services Computing, IEEE International Conference on*, 0:372–379, 2009. 20

[53] Johannes Köbler, Uwe Schöning, and Jacobo Torán. *The graph isomorphism problem: its structural complexity*. Birkhauser Verlag, Basel, Switzerland, Switzerland, 1993. 62

[54] Julius Koepke and Hannes Hannig. Performance evaluation of declarative annotation-based matching and mapping vs. lifting/lowering transformations. Technical report, Alpen Adria Universität Klagenfurt, 2011. 50

[55] Boris Konev, Carsten Lutz, Dirk Walther, and Frank Wolter. Logical difference and module extraction with cex and mex. In *Proceedings of the 21st International Workshop on Description Logics (DL2008)*, volume 353 of *CEUR-WS*, 2008. 64, 71, 72

[56] Jacek Kopecký, Tomas Vitvar, Carine Bournez, and Joel Farrell. Sawsdl: Semantic annotations for wsdl and xml schema. *IEEE Internet Computing*, 11(6):60–67, 2007. 3, 11

[57] Julius Köpke and Johann Eder. Semantic annotation of xml-schema for document transformations. In Robert Meersman, Tharam Dillon, and Pilar Herrero, editors, *Proc. of OTM'10 Workshops*, volume 6428 of *LNCS*, pages 219–228. Springer, 2010. 3, 11, 143

[58] Julius Köpke and Johann Eder. Semantic invalidation of annotations due to ontology evolution. In *OTM Conferences (2)*, pages 763–780, 2011. 3, 119, 143, 144

[59] Julius Köpke and Johann Eder. Logical invalidations of semantic annotations. In *To appear in Proc. CAiSE'12*, Gdansk, Poland, 6 2012. Springer. 3, 97, 144

[60] Markus Krötzsch, Frantisek Simancik, and Ian Horrocks. A description logic primer. *CoRR*, abs/1201.4089, 2012. 10

[61] Jianguo Lu, Ju Wang, and Shengrui Wang. Xml schema matching. *International Journal of Software Engineering and Knowledge Engineering*, 17(5):575–597, 2007. 27

[62] Luong, H, Dieng-Kuntz, and R. A rule-based approach for semantic annotation evolution. *Computational Intelligence*, 23(3):320–338, August 2007. 87

[63] Jayant Madhavan, Philip Bernstein, and Erhard Rahm. Generic schema matching with cupid. In *In The VLDB Journal*, pages 49–58, 2001. 21

[64] Kuhanandha Mahalingam, Michael, and N. Huhns. Ontology tools for semantic reconciliation in distributed heterogeneous information environments. In *Intelligent Automation and Soft Computing*, 1999. 2

[65] Deborah L. McGuinness, Richard Fikes, James Hendler, and Lynn Andrea Stein. Daml+oil: An ontology language for the semantic web. *IEEE Intelligent Systems*, 17:72–80, 2002. 62

[66] Eric Miller and Frank Manola. RDF primer. W3C recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-rdf-primer-20040210/. 1, 3, 11, 48

[67] Marvin Minsky. A framework for representing knowledge. Technical report, Cambridge, MA, USA, 1974. 8

[68] M. Missikoff, F. Schiappelli, and F. Taglino. A controlled language for semantic annotation and interoperability in e-business applications. In *Proc. of ISWC-03*, pages 1–6, 2003. 19

[69] Michele Missikoff and Francesco Taglino. Semantic mismatches hampering data exchange between heterogeneous web services. In *W3C Workshop on Frameworks for Semantics in Web Services*, 2005. 20, 38

[70] Boris Motik. On the properties of metamodeling in owl. In *In 4th Int. Semantic Web Conf. (ISWC 2005*, pages 548–562, 2005. 58

[71] Natalya F. Noy and Mark A. Musen. Promptdiff: a fixed-point algorithm for comparing ontology versions. In *Eighteenth national conference on Artificial intelligence*, pages 744–750, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence. 62, 71, 72, 73, 79

[72] Natalya Fridman Noy and Mark A. Musen. Prompt: Algorithm and tool for automated ontology merging and alignment. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 450–455. AAAI Press / The MIT Press, 2000. 62

[73] Natalya Fridman Noy and Mark A. Musen. Specifying ontology views by traversal. In *International Semantic Web Conference*, pages 713–725, 2004. 120

[74] Institute of Electrical and Electronics Engineers. *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*. IEEE, New York, 1990. 7

[75] W3C OWL Working Group. *OWL 2 Web Ontology Language: Document Overview*. W3C Recommendation, 27 October 2009. Available at `http://www.w3.org/TR/owl2-overview/`. 9, 98

[76] Raúl Palma, Peter Haase, Óscar Corcho, and Asunción Gómez-Pérez. Change representation for owl 2 ontologies. In Rinke Hoekstra and Peter F. Patel-Schneider, editors, *OWLED*, volume 529 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008. xiii, 66, 67, 71, 72, 73

[77] B. Parsia, E. Sirin, and A. Kalyanpur. Debugging owl ontologies. In *Proc. 14th International Conf. World Wide Web*, pages 633–640. ACM Press, 2005. 99, 114, 116

[78] Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean. SWRL: A semantic web rule language combining OWL and RuleML. W3C submission, W3C, May 2004. http://www.w3.org/TR/2008/REC-rdf-sparql-XMLres-20080115/. 30, 50, 70

[79] Mikalai Yatskevich Pavel Shvaiko, Fausto Giunchiglia. *Semantic Matching with S-Match*, volume Part 2, pages 183–202. 2010. 21, 63, 71, 72

[80] Peter Plessers and Olga De Troyer. Ontology change detection using a version log. In Yolanda Gil, Enrico Motta, V. Richard Benjamins, and Mark A. Musen, editors, *International Semantic Web Conference*, volume 3729 of *Lecture Notes in Computer Science*, pages 578–592. Springer, 2005. xiii, 65, 71, 72

[81] Lucian Popa, Yannis Velegrakis, Mauricio A. Hernández, Renée J. Miller, and Ronald Fagin. Translating web data. In *Proceedings of the 28th international conference on Very Large Data Bases*, VLDB '02, pages 598–609. VLDB Endowment, 2002. 22

[82] Eric Prud'hommeaux and Andy Seaborne. SPARQL query language for RDF. W3C recommendation, W3C, January 2008. http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/. 49

[83] Li Qin and Vijayalakshmi Atluri. Evaluating the validity of data instances against ontology evolution over the semantic web. *Information and Software Technology*, 2008. 131

[84] Erhard Rahm. Towards large-scale schema and ontology matching. In Zohra Bellahsene, Angela Bonifati, and Erhard Rahm, editors, *Schema Matching and Mapping*, Data-Centric Systems and Applications, pages 3–27. Springer Berlin Heidelberg, 2011. 10.1007/978-3-642-16518-41. 21, 22

[85] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, 2001. 14, 19, 21, 22, 23, 24

[86] R. Reiter. A theory of diagnosis from first principles. In Joerg Siekmann, editor, *8th International Conference on Automated Deduction*, volume 230 of *Lecture Notes in Computer Science*, pages 153–153. Springer Berlin / Heidelberg, 1986. 10.1007/3-540-16780-387. 116

[87] Khalid Saleem and Zohra Bellahsene. Complex schema match discovery and validation through collaboration. In *Proceedings of the Confederated International Conferences, CoopIS, DOA, IS, and ODBASE 2009 on On the Move to Meaningful Internet Systems: Part I*, OTM '09, pages 406–413, Berlin, Heidelberg, 2009. Springer-Verlag. 25, 26

[88] Khalid Saleem, Zohra Bellahsene, and Ela Hunt. Porsche: Performance oriented schema mediation. *Inf. Syst.*, 33:637–657, November 2008. 26

[89] Stefan Schlobach and Ronald Cornet. Non-standard reasoning services for the debugging of description logic terminologies. In Georg Gottlob and Toby Walsh, editors, *IJCAI*, pages 355–362. Morgan Kaufmann, 2003. 99, 113, 116

[90] Sana Sellami, Aicha-Nabila Benharkat, Rami Rifaieh, and Youssef Amghar. Advanced internet based systems and applications. chapter Extension of Schema Matching Platform ASMADE to Constraints and Mapping Expression, pages 223–234. Springer-Verlag, Berlin, Heidelberg, 2009. 27

[91] Sana Sellami, Nabila Benharkat, Rami Rifaieh, and Youssef Amghar. Schema Matching for Document Exchange: A Constraint Based Approach. In *THE INTERNATIONAL*

CONFERENCE ON SIGNAL-IMAGE TECHNOLOGY & INTERNET-BASED SYSTEMS (SITIS' 2006), pages 299–309. springer Verlag, LNCS series, December 2006. 27

[92] Pavel Shvaiko and Jérôme Euzenat. A survey of schema-based matching approaches. In Stefano Spaccapietra, editor, *Journal on Data Semantics IV*, volume 3730 of *Lecture Notes in Computer Science*, pages 146–171. Springer Berlin / Heidelberg, 2005. 10.1007/116034125. 21, 22

[93] Ljiljana Stojanovic, Alexander Maedche, Boris Motik, and Nenad Stojanovic. User-driven ontology evolution management. In *EKAW '02: Proceedings of the 13th International Conference on Knowledge Engineering and Knowledge Management. Ontologies and the Semantic Web*, pages 285–300, London, UK, 2002. Springer-Verlag. 60, 70, 72

[94] Nenad Stojanovic, Alexander Maedche, Steffen Staab, Rudi Studer, and York Sure. Seal - a framework for developing semantic portals. In *K-CAP 2001 - First Intenational Conference on Knowledge Capture, Victoria, Canada, Oct. 21-23, 2001*. ACM, 2001. 60

[95] Marcin Szymczak. Semantic annotation-based xml document transformation. Masterthesis, Alpen Adria Universität Klagenfurt, Universitätsstrasse 65-67, 9020 Klagenfurt, September 2010. 44

[96] Marcin Szymczak and Julius Koepke. Matching methods for semantic annotation-based xml document transformations. In *To appear in Proceedings of the IWIFSGN'2011, Warsaw*, September 2011. 3, 44, 143

[97] Victoria Uren, Philipp Cimiano, José Iria, Siegfried Handschuh, Maria Vargas-Vera, Enrico Motta, and Fabio Ciravegna. Semantic annotation for knowledge management: Requirements and a survey of the state of the art. *Web Semantics: Science, Services and Agents on the World Wide Web*, 4(1):14 – 28, 2006. 11

[98] M. Vujasinovic, N. Ivezic, B. Kulvatunyou, E. Barkmeyer, M. Missikoff, F. Taglino, Z. Marjanovic, and I. Miletic. Semantic mediation for standard-based b2b interoperability. *Internet Computing, IEEE*, 14(1):52 –63, jan.-feb. 2010. 5, 19, 43

[99] Evan K. Wallace and Christine Golbreich. OWL 2 web ontology language new features and rationale. Technical report, W3C, October 2009. http://www.w3.org/TR/2009/REC-owl2-new-features-20091027/. 57

[100] Priscilla Walmsley and David C. Fallside. XML schema part 0: Primer second edition. W3C recommendation, W3C, October 2004. http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/. 3, 7

[101] Hai H. Wang, Natasha Noy, Alan Rector, Mark Musen, Timothy Redmond, Daniel Rubin, Samson Tu, Tania Tudorache, Nick Drummond, Matthew Horridge, and Julian Seidenberg. Frames and OWL Side by Side. In *9th International Protégé Conference*, 2006. 57

[102] An Yuan, Alex Borgida, and John Mylopoulos. Discovering and Maintatining Semantic Mappings between XML Schemas and Ontologies. *Journal of Computer Science and Engeneering*, 5:1–29, December 2007. 19, 20