Diploma Thesis

February 20, 2008

RDF Graph Transformation

Bridging between Ontologies

Matthias Hert
of Messen SO, Switzerland (02-908-879)

supervised by
Prof. Dr. Harald Gall
Dr. Gerald Reif

University of Zurich
Department of Informatics

s.e.a.l.
software evolution & architecture lab

Diploma Thesis

RDF Graph Transformation

Bridging between Ontologies

Matthias Hert

University of Zurich
Department of Informatics

s.e.a.l.
software evolution & architecture lab

Diploma Thesis

Author: Matthias Hert, mhert@access.uzh.ch

Project period: August 27, 2007 - February 27, 2008

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Acknowledgements

First of all, I want to express my gratitude to Gerald Reif for his support and the many helpful discussions we had. He always had time for me and my issues. I was also able to benefit from his broad knowledge of Semantic Web technology and the many inputs he gave me.

Likewise, I want to thank Professor Harald Gall for giving me the opportunity to carry out my thesis within his research group.

I am also grateful to Andy Seaborne for helping me with my not so common problems related to Jena, ARQ, and SPARQL. His answers to my questions were always quick and extensive.

Last but not least, many thanks go to my parents who supported me not only during the thesis but throughout my entire studies. I thank my brother as well, not only for proofreading the written part of my thesis but also for everything else.

Abstract

The Semantic Web uses Web page annotations to enable machines to access the semantics of the pages content. This is done by the use of the data representation languages RDF and OWL to define ontologies for specific application domains. By the decentralized organization of the Web, it is not feasible to expect that only one ontology for each domain will be defined and used by everyone. There already are multiple ontologies that relate to the same or overlapping domains and this will not change in the future. Thereby, it is getting more and more difficult for developers of Semantic (Web) applications to support all these existing and future vocabularies. This problem could be solved with the help of a transformation service that can map between different ontologies from related domains. With it, applications would no longer need to understand unknown ontologies as they could inquire the transformation service to exchange data with other applications that use different ontologies.

In this thesis, we present an approach to such a transformation service that performs transformations based on mapping definition files expressed in a simple and easy to understand XML syntax. We introduce a flexible mapping language that is applicable to a wide variety of transformation situations. In addition, we provide a prototype implementation of such a service that understands the proposed mapping language and therefore demonstrate its feasibility.

The contributions of this thesis are the general applicable mapping language expressed as an XML application and the prototype implementation of the transformation service that understands such mappings and uses them to translate between different ontologies. Thereby, it becomes feasible to create many different mappings in an easy to use definition language. To further demonstrate the value of our transformation service, we integrate it into an application, the Semantic Clipboard, that is able to copy semantic data from Web pages to applications which use different ontologies.

Zusammenfassung

Das Semantic Web bietet grosse Chancen um die Semantik von Daten in einer Form zu annotieren die von Maschinen direkt verarbeitet werden kann. Im Semantic Web werden Sprachen wie RDF und OWL verwendet um Ontologien für spezifische Anwendungsdomänen zu definieren. Durch die dezentralisierte Organisation des Webs ist es nicht plausibel zu erwarten, dass nur eine einzige Ontologie für jede Domäne definiert und diese von allen verwendet wird. Es gibt bereits heute mehrere Ontologien, die sich auf dieselbe oder überlappende Domänen beziehen, was sich auch in Zukunft nicht ändern wird. Deshalb wird es für Entwickler von semantischen (Web-) Anwendungen immer schwieriger, alle diese jetzt und in Zukunft existierenden Vokabulare zu unterstützen. Dieses Problem könnte mit einer Zwischenschicht in Form eines Transformationsservices gelöst werden, welcher verschiedene Ontologien aus verwandten Domänen aufeinander abbilden kann. Damit werden Anwendungen nicht mehr länger alle fremden Ontologien verstehen müssen, da sie einfach den Transformationsservice anfragen können um Daten mit Anwendungen auszutauschen, die eine andere Ontologie verwenden.

In dieser Arbeit präsentieren wir einen Ansatz für solch einen Service der Transformationen durchführt, welche auf Mapping-Definitionsdateien in einer einfach zu verstehenden XML Syntax basieren. Wir führen eine flexible Mapping Sprache ein, die in einer Vielzahl von Transformationssituationen einsetzbar ist. Zusätzlich implementieren wir einen Prototyp dieses Services, welcher die vorgeschlagene Mapping Sprache versteht und zeigen dadurch die Machbarkeit dieser Mappings.

Die Ergebnisse dieser Arbeit sind die allgemein anwendbare Mapping Sprache, definiert als eine XML Anwendung und der Prototyp des Transformationsservices, welcher diese Mappings versteht und verwendet um zwischen verschiedenen Ontologien zu übersetzen. Dadurch wird es möglich viele Mappings in einer einfach zu benutzenden Definitionssprache zu erstellen. Um den Nutzen unseres Transformationsservices weiter aufzuzeigen, integrieren wir ihn in eine Anwendung, das Semantic Clipboard, welche es ermöglicht semantische Daten von Webseiten in Anwendungen zu kopieren welche unterschiedliche Ontologien benutzen.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

1.1 Semantic Web Overview

This section introduces the basic principles and mechanisms of the Semantic Web. Starting from today's Web it shows the shortcomings that lead to the idea of the Semantic Web.

1.1.1 Today's Web and the Semantic Web Vision

The World Wide Web has changed the personal and the business world. The way people communicate with each other and conduct business has changed revolutionary. This transformation has also changed the role of computers from their original purpose as large numerical calculators to information processors, like as databases, text editing systems, and games. With the current developments, there is another transition happening that puts computers towards the role as entry points to the information highways [AvH04].

The current Web is focusing on the human user and therefore delivers its content in human readable form, even if it is generated automatically from databases. Unfortunately, this form of presentation is poorly suited for processing by software tools. With the ever increasing amount of information from more and different sources, it gets harder to find the intended information for human users as well. This in turn creates the need for support from software tools, which exists today in the form of keyword-based search engines such as Google and Yahoo. However, their use exhibits some serious problems:

High recall, low precision Even if the main relevant pages are retrieved, they are of little use if thousands other mildly relevant or irrelevant documents were also retrieved.

Low or no recall It happens that important and relevant pages are not retrieved or that in an extreme case we get no results back at all.

Results are highly sensitive to vocabulary Often we retrieve not the pages we wanted with the keywords we initially selected, because the relevant documents use different terminology.

Results are single Web pages If our query matches the important and relevant documents, we only retrieve complete Web pages containing the wanted information. We still have to manually extract the desired parts of information, probably from multiple pages and put them together.

Therefore, a human user is needed to compose an adequate query and to browse several result pages to retrieve the actual information in demand. Despite the improvements in search engines, artificial intelligence, and computational linguistic these problems could not been solved

so far. The Semantic Web initiative lead by Tim Berners-Lee and the World Wide Web Consortium (W3C) tries to solve these problems with a different approach. The basic idea is to represent the Web content in a more machine-processable form instead of trying to understand the human-readable data. Then, software can take advantage of this new representation by using intelligent techniques to incorporate the semantics of data [AvH04].

It is important to understand that the Semantic Web initiative has not the intention to build a separate new Web; it will only extend the current Web with explicit metadata for easier machine processing.

1.1.2 Explicit Metadata

As the term *semantic* in Semantic Web suggest, the metadata is concerned with the meaning of an entity, its semantics. It builds a relationship between the object represented in the information system and the real world object. In order to extract this metadata without the need for sophisticated techniques from artificial intelligence and natural language processing, it must be defined in an explicit form additionally to the original data. In the Semantic Web, this explicit metadata is expressed in RDF, which is introduced in Section 1.1.4 [AvH04].

1.1.3 Ontologies

The term *ontology* originates from philosophy, where it names the subfield concerned with the study of the nature of existence. However, in recent years computer science has hijacked this word as many others before and has given it a different technical meaning [AvH04]. T.R. Gruber defines an ontology as an explicit and formal specification of a conceptualization [Gru93].

In general, an ontology describes formally a domain of discourse. These descriptions typically consist of terms denoting important concepts (classes of objects) of a domain and relationships between these terms. The types of these relationships are manifold with the subclass relationship as one of the most important. The subclass relationship is defined as follows: a class C is a subclass of another class C′, if every object in C is also an object of C′. Therewith, it is possible to define entire hierarchies of classes. Apart from subclasses, relationships can take the form of properties, value restrictions, disjointness statements, and specification of logical relationships between objects [AvH04].

In the context of the Web, ontologies provide a shared understanding of a domain, which is necessary to overcome differences in terminology. Ontologies can also be used to improve the situation in the keyword-based search engines example presented in Section 1.1.1. Instead of just searching for a keyword, an engine based on ontologies can search for specific concepts. If it then fails to find any relevant documents or the responses are too numerous, it can exploit the generalization/specialization information given in the hierarchies to broaden/narrow the search [AvH04].

1.1.4 Architecture of the Semantic Web

Layered architectures are a widespread principle in computer science to decompose a complex problem into smaller and less complex subproblems. Each higher layer relies on the services offered by the lower layer and provides services for the next higher layer. Another advantage of this architecture is that it is easier to achieve consensus on small steps independently on every layer than on the complete problem. This is necessary, because the Semantic Web still is a relatively young research field and therefore standards and common agreement just start to emerge.

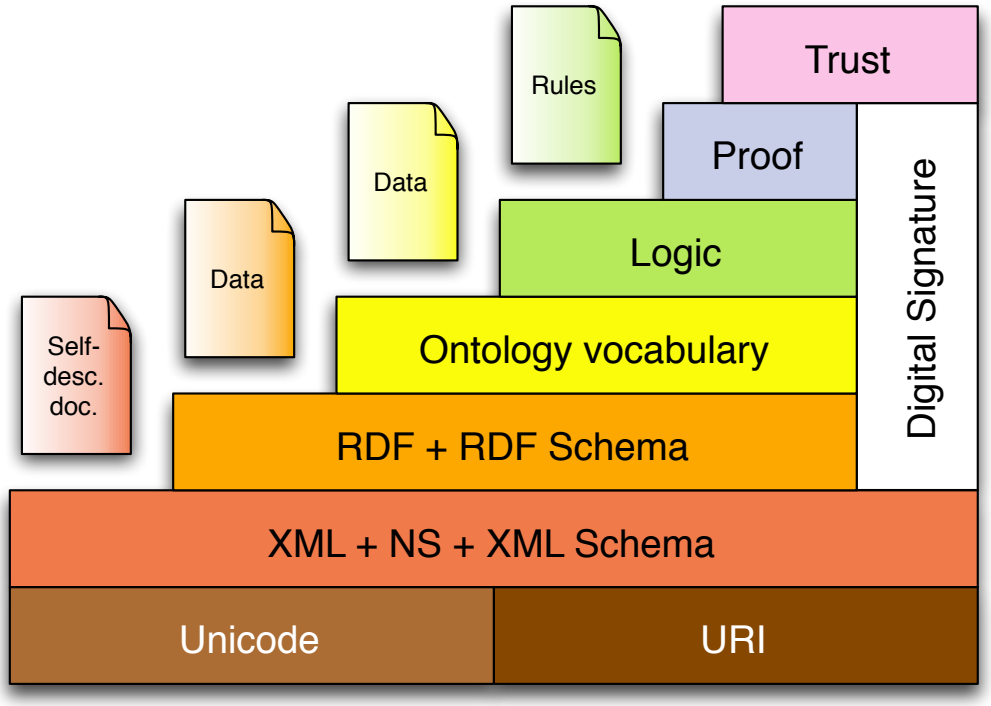

Figure 1.1: A layered approach to the Semantic Web [AvH04]

The Semantic Web obeys such a layered architecture too as shown in Figure 1.1. This section introduces the topics of each layer briefly [AvH04].

Basic Layers

The two lowermost layers are not exclusive Semantic Web technologies. They appear in many fields of computer science and especially the World Wide Web community. Therefore, they are in widespread use and there exist many tools to work with these technologies.

Unicode is a character encoding system. Before Unicode was developed, there were countless systems covering only specific parts of what Unicode supports today. For example, one encoding system would only contain the English characters but not the additional letters of the German and French alphabet. Unicode solves this problem by providing a unique number for every character independent of language, platform, and program [Uni07].

A *Uniform Resource Identifier (URI)* is a compact sequence of characters that identifies an abstract or physical resource. They are a generalization of the Universal Resource Locator (URL) familiar from the current Web. URLs not only identify resources on the Web, they provide a means of locating the resource by describing its primary access mechanism [BLFM05]. Resources are a main part of the Semantic Web, so we need a way to identify them. Common URLs are not well suited for this purpose, because in the Semantic Web we want to describe all kinds of resources, including those that cannot be accessed through the Web like, for instance, persons. In that case, a URL would suggest that we can retrieve the person itself from the Web and not only information about that person. Therefore, URIs are used to uniquely identify resources on the Semantic Web.

The *eXtensible Markup Language (XML)* uses tags to add markup to arbitrary text documents.

The term 'extensible' means that there is no predefined set of tags in XML in contrast to HTML. This makes XML practical as a language for expressing explicit metadata as well as for modeling ontologies [BPSM+06]. Regarding the Semantic Web, XML is surely suitable as an exchange format for metadata due to its wide distribution and its ties to the Web.

XML namespaces provide a simple method for qualifying element and attribute names used in XML documents by associating them with namespaces expressed as URI references [BHLT06]. XML namespaces and concepts based on them find applications throughout the Semantic Web technologies.

XML Schema provides a means for defining the structure, content, and semantics of XML documents. It allows machines to check automatically whether a given XML document conforms to a set of rules or not [SMT00]. As we will see later in this section, XML Schema is not powerful enough to define the structure and semantics of the statements in the upper layers of the Semantic Web architecture, but it is still useful when dealing with XML representations and as supplier for common data types.

RDF and RDF Schema Layer

XML is indeed a suitable language for defining and exchanging data and metadata, but it provides no means to define the semantics of this (meta-)data. In XML, it is possible to express the same facts in different serializations, which causes trouble when sharing data. To resolve this problem, the Semantic Web needs a standardized way to describe metadata. This is done by the *Resource Description Framework (RDF)*, which represents the data model of the Semantic Web. The basic building block of RDF is the statement composed of a subject-predicate-object triple, which provides a means for describing arbitrary resources. The subject can be any resource identifiable by an URI, while the predicate defines a property of a resource and is identified with an URI itself. The object, on the other hand, represents the value of that property and can be either a literal or a resource identified again by an URI. As RDF itself is only an abstract data model, it needs a concrete syntax for use in real world applications. There is not one exclusive syntax but several different ones, which are useful in various circumstances. Probably the most popular serialization is the XML format called RDF/XML. It is best suited as a form for RDF to exchange metadata with other Semantic Web tools over the Web. As XML has the reputation of being somewhat bloated there exist other popular syntaxes like, for instance, N3 which is more compact than XML and easy to read for human users as well. The important aspect is that despite all these different syntactical representations the underlying model of the Semantic Web are triples forming a graph [AvH04]. This fact is covered in more detail in Section 1.2.

RDF is domain-independent, that means there is no predefined terminology for any domain. The users have to define the desired vocabulary themselves in the schema language *RDF Schema (RDFS)*. The name RDF Schema is confusing, because RDFS defines the vocabulary and not the structure of documents, like XML Schema for example. RDFS itself is built on RDF triples, which means every RDFS document is also an RDF document. RDFS supports the definition of classes, properties with domain and range restrictions, and the concept of inheritance to form hierarchies of classes and properties. This describes the most important capabilities of RDFS, which shows that its expressiveness is too limited to create complex ontologies. Therefore, the Semantic Web needs a more expressive ontology language to address this issue, which is presented on the next higher layer: the ontology layer [AvH04].

Ontology Layer

In today's Semantic Web, the preferred ontology definition language is the *Web Ontology Language (OWL)*. It adds more capabilities for describing classes and properties along with support for

reasoning and formal semantics. In contrast to RDF and RDFS, it supports additionally:

Relationships between classes In RDFS we can only state the subclass relationship between two classes, but sometimes we wish to express different relationships such as disjointness between classes. OWL gives us a wide number of such inter-class relations.

Boolean combination of classes In some cases, it would be practical to define a new class by combining other classes using Boolean combinations. While not possible with RDFS, OWL enables us to define new classes based on the union, intersection, or complement of two or more other classes.

Local scope of properties If we define the domain or range of a property in RDFS it applies globally to all classes and thereby restricts the use of a property. In OWL, it is possible to define the scope of such restrictions in a local and more detailed manner.

Cardinality restrictions OWL allows us to express exact cardinality restrictions on properties, whereas this is not possible in RDFS at all.

Special characteristics of properties Sometimes it is useful to say that a property is transitive, unique, or the inverse of another property. Again, RDFS gives us no means to do so, but OWL does.

Ideally, to be consistent with the layered architecture of the Semantic Web, OWL would be an extension of RDF Schema. It would use the RDF meaning for classes and properties and would only add language primitives to support the richer expressiveness. Unfortunately, this would lead to uncontrollable computational properties and hinder efficient reasoning. Therefore, the W3C Ontology Working Group decided to define OWL as three different, decreasingly expressive sublanguages:

OWL Full The entire language is called OWL Full. It uses all the OWL primitives and allows their combination in arbitrary ways with primitives from RDF and RDFS. The advantage of OWL Full is that it is fully upward compatible with RDF, both syntactically and semantically. The disadvantage is that OWL Full is so powerful that it is undecidable, making a complete or efficient reasoning support unfeasible.

OWL DL In order to regain computational efficiency, OWL DL (which stands for Description Logic) is a sublanguage of OWL Full that restricts the use of OWL constructors to each other. Thus, OWL DL corresponds to a well-studied description logic and therein lies its advantage: efficient reasoning support. The disadvantage is that we lose full compatibility with RDF. Not every RDF document will be a legal OWL document unless it is extended in some ways and restricted in others.

OWL Lite OWL Lite is a further restriction of OWL DL. It excludes several language constructors like arbitrary cardinality and disjointness statements. Compared to the disadvantage of this restricted expressiveness stands the advantage of a language that is easier to grasp for users and easier to implement for tool builders.

It is important to note that all three sublanguages are upward compatible, which means every legal OWL Lite ontology is a legal OWL DL ontology and every legal OWL DL ontology is a legal OWL Full ontology. The same holds true for valid conclusions from the single sublanguages. Likewise, every legal OWL document, regardless in what sublanguage it is written, is a legal RDF document. Summarized, OWL enables us to define complex ontologies and with the help of a logic reasoner to derive even more (implicit) information from given ontologies [AvH04].

Top Layers

The top three layers *Logic*, *Proof*, and *Trust* are not yet in use as much as the other layers and they are currently rather the subject of research than widespread application. The Logic layer enhances the ontology language further and allows the writing of application-specific declarative knowledge. The Proof layer involves the actual deductive process, the proof representation and validation. In combination with digital signatures, the Trust layer deals with security of operations and the quality of the provided information for Semantic Web users [AvH04].

1.2 RDF as a Graph

A directed Graph G is defined as $G = (V, E)$ and consists of a set $V = \{1, 2, ..., |V|\}$ of vertices (also called nodes) and a set $E \subseteq V \times V$ of edges (also called arcs). A pair $(v, v') \in E$ is called an edge from v to v', with v named the head and v' the tail. [OW02]

The basic building block of RDF is a triple consisting of a subject, a predicate, and an object. The predicate connects the subject with the object and therefore a triple can as well be represented as a node-arc-node link as depicted in Figure 1.2. Given the definition of a directed graph, such a triple can be seen as a simple directed graph as well. If we extend this view to multiple triples belonging together, we still get a graph, although a bigger and more complex one. As any RDF expression has a collection of triples as its underlying structure, we can understand any RDF document as a directed graph [KC04].

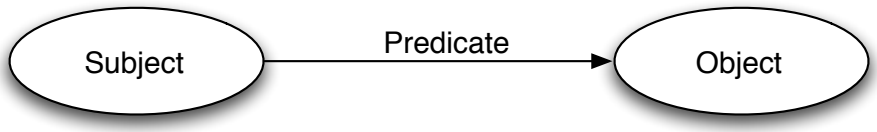

Figure 1.2: An RDF triple as a directed graph

Let us illustrate this association between statements and graphs in a more concrete example. Assume the following statements are given: "There is a Person identified by `http://www.w3.org/People/EM/contact#me`, whose name is Eric Miller, whose email address is `em@w3.org`, and whose title is Dr.". From these statements, we can derive the respective RDF triples and they could be represented in the form of the RDF graph in Figure 1.3. This graph visualizes the RDF triples in an abstract form independent of any specific syntax. The subjects are displayed as green ovals with their URI as text written inside them. If the objects represent URIs too, they are shown likewise as green ovals, but if the represent literals they are depicted as yellow rectangles with their values written inside the box. At last, the predicates are drawn as arrows connecting the respective subject with its object and the full URI written next to the arc [MM04].

1.3 Ontology Alignment and Mapping

Given that the Semantic Web is only an extension of the current Web, this *new Web* shares a lot of its properties with the *old Web*. This is especially true in terms of distribution and heterogeneity. Therefore, it is neither possible nor desirable to create a central control to supervise all development of the (Semantic) Web. This implies that one of the main problems of the Semantic Web is the integration of resources. There will be multiple ontologies for the same domain, defined by different actors and the individual ontologies themselves will become heterogeneous over time with

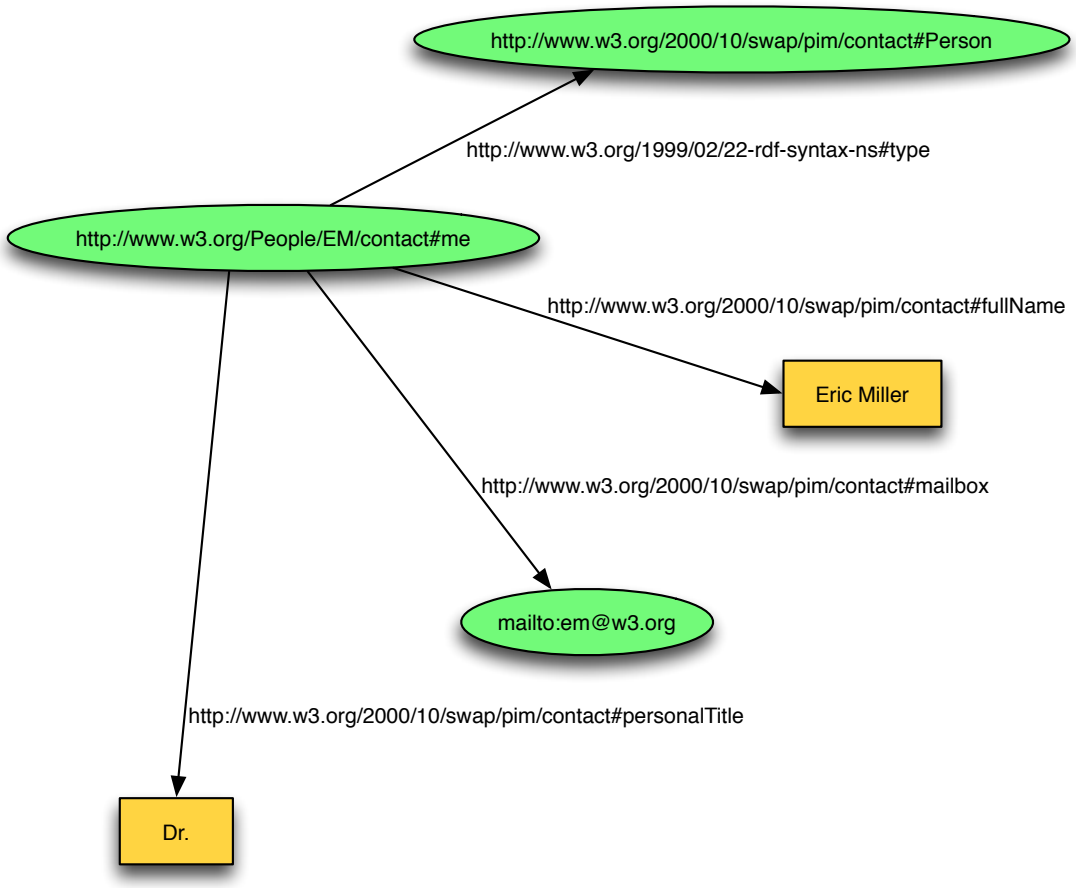

Figure 1.3: An example of an RDF graph [MM04]

the introduction of new and probably incompatible versions. The emerging of Web and ontology standards cannot solve this problem completely. There will always be user groups who either do not want to use a standard ontology or cannot use it, because it does not suit their problem in an appropriate fashion. Therein lies the need for ontology alignment and possibilities to bridge between multiple ontologies. The process of such an ontology alignment is defined as finding relationships between entities belonging to two different ontologies. This alignment can either be calculated automatically or defined manually through an expert. After this process is completed, it results in a mapping that can be used to translate between the involved vocabularies or merge them into a new, integrated ontology [Euz04].

1.3.1 Use Cases

Let us now look at a set of use cases to fortify the importance of ontology alignment and reveal possible practical applications.

Agent Communication

Agents are autonomous computer entities with the ability to interact with other agents. For this interaction, they need to communicate with each other and they need a way to express their intentions which can be done through ontologies. If two autonomous and independently designed agents meet, they will be able to exchange messages but are unlikely able to understand each other, due to the use of different ontologies. To solve this communication problem, an ontology alignment is needed. This can be realized in several ways, for example in the form of a translation service offering either direct mappings between different ontologies or mappings from the agents ontologies into a common standard ontology [Euz04].

Web Service Integration

The identification of Web Services that fulfill a given requester goal is the idea behind automatic Web Service discovery. Both the requester goal and the Web Service capability must be described in a declarative and machine-processable way. An all-embracing global ontology will not be feasible in this case either, so the goal and capabilities are most likely specified in different and domain-specific ontologies. To enable a maximum degree of interoperability, a mechanism is required to mediate between the various ontologies, in other words a mapping is needed [Euz04].

Catalog Matching

E-Commerce businesses rely on their electronic catalog to describe the goods on sale. Through the independence of these enterprises, almost every catalog will have its own format. As long as the catalog needs only to be used by the original firm this causes no major problems. Not until the company wants to sell its goods on an e-Marketplace. The operator of this electronic marketplace dictates his own catalog format, which leaves the interested company with two options. It can replace its internal catalog format with the one from the marketplace firm, but this is a lot of work and probably a constraint too. The other option is to create a second catalog in the format for the e-Marketplace, which means a doubling in work to keep both catalogs up to date. This problem increases even more if the company wants to participate in multiple e-Marketplaces. We can evidently see the need for an automatic and ideally dynamic mapping here as well [Euz04].

Information Retrieval from Heterogeneous Multimedia Databases

Multimedia documents like video, audio, and still images have long been stored in databases for building digital archives. Together with corresponding metadata, it enables easier preservation and access to this data for the holding organization. With advances in data networks and Web technologies, these archives open up towards the Web and its users. The original databases were designed and built independently by the content holding organizations, so were the respective formats of the metadata. The new users of such databases are now confronted with different metadata ontologies, which makes retrieval of specific content hard and costly. An agreement among all content providers to convert their metadata into a single format is hardly feasible, hence we need to bridge this gap. This can be done with a mediator translating between the requester and the different data sources [Euz04].

P2P Information Sharing

Peer-to-Peer (P2P) systems have recently received a lot of attention, particularly in the form of P2P file sharing systems. These systems support only few options to describe the file contents. Either they provide a simple schema that is shared among all peers and cannot be changed locally

by one party or they offer no possibility at all to store additional data about a file, except maybe within the filename. While this may sound tolerable in the case of simple files, if we extend this concept to general information, it becomes clear that a more sophisticated annotation mechanism will be needed. Every peer on the network should be able to define freely the annotations for his piece of information. However, this would lead again to heterogeneous metadata and therefore an alignment system must be created to gain the full benefit of this additional data [Euz04].

Personal Information Delivery

Internet radio has become a reality by now, but at the moment it is not really different from conventional radio concerning the content of the radio program. Although we can easily receive more radio channels then ever before, we do not have more opportunities to influence the programming. A smart internet radio analyzes the music taste of its listeners and selects an appropriate audio stream. To achieve this goal the internet radio must detect the interests of each user. This seems to be easy through filtering based on music genres, but the problem herein is that different people assign the same music to different genres and there is no common agreement on the correct classification. This problem relates to ontology alignment due to the fact that the internet radio must translate between the styles of music categorization of the users and the content providers [Euz04].

1.3.2 Semantic Heterogeneity

In a distributed and open system like the Semantic Web, heterogeneity cannot be avoided. Different actors have different backgrounds, intentions, and use different tools to define an ontology. These are reasons why heterogeneity emerges even if we try to be as objective as possible while constructing an ontology. The differences can appear in different forms at various levels, therefore we use this section to classify them into four main levels [Bou05].

The Syntactic Level

Heterogeneity at the syntactic level arises from the choice of the representation format. For instance, there are several languages for ontology representation like OWL and KIF, both based on their own syntax. Differences at this level are not limited to ontology research as they are common in computer science in general and therefore well understood [Bou05].

The Terminological Level

At the terminological level, we face all kinds of mismatches related to naming of entities in an ontology. There are multiple possible sub-levels of discrepancy imaginable. Two ontologies could mean the same entity but use synonymous words to represent it in their respective ontology. An even trickier problem occurs, if ontology engineers use one ambiguous word to name multiple different concepts. Another pitfall is the use of abbreviations in names or different, but legal spelling and similar syntactic variations. A last example is the use of words from different languages to describe things. Due to the different cultural backgrounds and varying experience among ontology engineers, this level of heterogeneity is widely spread [Bou05].

The Conceptual Level

At the conceptual level, all discrepancies are related to the content of an ontology. We can further subdivide these mismatches in two classes. First, there are epistemic differences that have

to do with the assertions ontology designers make about the chosen entities. Second, there are metaphysical differences that have to do with how the world is divided into parts, which means what concepts from the world are represented as entities, properties, or relations in the ontology. The practical forms in which these metaphysical differences can occur are innumerable, but in accordance with the artificial intelligence literature (in particular [BBG00]) on this topic, we cluster them in three abstract types. The first type is called *coverage* and describes the variations in ontologies as different parties decide to represent different portions of the world or even of a single domain. An example may be one sports ontology that includes car racing, whereas a second may ignore it completely as part of the sports domain. The second type is called *granularity* and concerns the level of detail an ontology exhibits. For instance, a tax ontology may only offer the generic concept of a document, while a library ontology may differentiate documents farther into books, articles, and so on. The last type is called *perspective* and can arise even if the other two types fit. It takes the form of different points of view on a domain. For example, two ontologies may represent the same domain and cover the same parts at the same granularity, but at different points in time. That means, the same property can hold in one time and not hold in the other. Unfortunately, mappings alone cannot solve all of the heterogeneities described at this level [Bou05].

The Semiotic/Pragmatic Level

Finally, discrepancies at the semiotic/pragmatic level have to do with different users of one ontology interpreting it in various contexts differently. For instance, if the concept of Europe appears in a multimedia ontology along the path *Images/B&W/Europe*, we should not conclude that it is equivalent to the concept of Europe in a geography ontology, because it most likely represents images taken in Europe in the first ontology and the continent of Europe in the second. This means that the structure of the classification can play an important role in the creation of mappings [Bou05].

1.4 Problem Statement

As already described in Section 1.3 the Semantic Web community has in some cases defined more than one ontology for similar or overlapping domains. It is also evident that these various ontologies will not be replaced with one unified ontology, but rather additional ontologies will be defined deteriorating the situation even more. This causes problems for applications that want to process this data. Not only would they have to know all of the currently available ontologies, but they would also need to be modified every time a new ontology is defined. To overcome these problems, a service is needed that translates RDF graphs from a source ontology into a target ontology and vice versa. With this solution, an application that receives data expressed in an unknown ontology can send this data together with the identifier of the desired target ontology to this service and gets back the translated data. The application then can understand and therefore process this data. If the original data came from another software program and the processing application returns the result in its ontology, it may happen that the original application does not understand the translated data. In that case, it needs to use the transformation service as well to translate the data back in an understandable vocabulary. Hence, to get the most value out of a mapping, the transformation service must derive transformations from the source to the target ontology and the opposite way from the target to the source.

In this thesis, we define an ontology transformation language that enables the definition of mappings from one ontology into another. Then, we implement a prototype of a transformation service that will generate and apply transformations from this mapping definition that work in

both directions; this means transformations forwards from the source into the target ontology and backwards from the target into the source.

1.5 Thesis Overview

After the introduction to the Semantic Web and ontology alignment in this first chapter, the remainder of this thesis is structured as follows: Chapter 2 gives an overview of related work. It briefly describes other mapping approaches and how they represent their mappings. In Chapter 3 we gain the requirements for a mapping language. We do this by first analyzing the two application domains 'contact data' and 'event data'. The outcome of this will be a set of requirements that are used as the basis for the definition of our mapping language introduced in Chapter 4. There, we describe the general parts of a mapping and then deal with each requirement from the prior chapter. Chapter 5 is devoted to the architecture and major relationships of components in our prototype implementation, whereas Chapter 6 concentrates in more detail on the individual components and their implementation. In Chapter 7 follows an evaluation of the prototype and the mapping language with an example not considered in the previous chapters. Chapter 8 describes the integration of our transformation service into a real world application. This application is called the Semantic Clipboard and its design and implementation is presented in that chapter. Finally, Chapter 9 concludes this thesis with a short summary, the mention of the main results, and ideas for future work.

Chapter 2

Related Work

The principal task of the system developed during this thesis is the automatic translation between different, but related ontologies. For this purpose, we need to define an appropriate mapping. This must be represented and saved in a way that we can easily use it in the transformation process. Therefore, in the next section we take a look at how other system, with sometimes different primary goals, make up their mappings.

2.1 Representation of Mappings

Anchor-PROMPT

Anchor-PROMPT [NM01] is an ontology merging and alignment tool that can also be used to generate mappings between two ontologies. Based on a variety of methods it produces such mappings in a semi-automatic way. The calculation of the similarity between the entities happens automatically; the user only provides anchor-pairs at the beginning and accepts or rejects matches at the end of the comparison. The tool is implemented as a plugin for the ontology editor and knowledge acquisition system Protégé [Pro]. Therefore, it can use a lot of functionality from its host to meet the intended purpose. This means also that its representation of the mapping is aligned with the native formats of Protégé. Consequently, the users are not intended to see the actual format of the mapping as they are supposed to use the graphical controls to perform transformation and other operations on the ontologies. So any usage of this mapping representation outside the Protégé environment is not reasonable.

Glue

The goal of Glue [Doa02] is to semi-automatically find schema mappings for data integration. It uses machine learning techniques to find the appropriate matches. The final results are stored in a simple XML file with pairs of corresponding objects. Each pair consists of a source element with the item from the source schema and a mediated element with the matching item from the destination schema. From this representation, we can clearly see that the primary purpose of this system lies in the discovery of adequate mappings and not in the actual translation of data.

IF-Map

IF-Map [KS03] is a system for automatic ontology mapping. One of its key strength is the support for a wide base of ontology languages. For the further processing, IF-Map translates the input

ontology into Horn logic. One reason for this conversion lies in the support for multiple input languages and the resulting need for a common, but independent internal data format. The other and more important reason is based on the further handling of the data, that is the actual calculation of the alignment which is done completely in Prolog. The resulting mapping is likewise expressed in Prolog clauses but can be exported to an RDF/XML format for further usage, especially on the Web. In this RDF/XML representation an URI is associated with every match found during the mapping process. This URI serves as the subject in RDF triples defining the mapping and the elements from the source and destination ontologies are the objects. Listing 2.1 shows an excerpt from a sample mapping.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:NS0="http://ecs.soton.ac.uk/~yk1/">
  <rdf:Description rdf:about="http://ecs.soton.ac.uk/~yk1/Infomorph4">
    <NS0:type>concept</NS0:type>
    <NS0:fromRefOnto>document</NS0:fromRefOnto>
    <NS0:toLocalOnto>publication</NS0:toLocalOnto>
  </rdf:Description>
</rdf:RDF>
```

Listing 2.1: Excerpt from sample IF-Map mapping [KS03]

With this representation it is possible to import such mappings back into IF-Map or to use the won mapping information, probably in different ways, with other Semantic Web applications.

RDFTranslator

RDFTranslator [Kru] is part of the MarcOnt initiative [Mar], which provides a set of tools for collaborative ontology development. One of this tools is RDFTranslator with the sole purpose of translating RDF documents from one ontology to another, similar to what XSLT achieves for XML documents. The RDFTranslator takes as input an RDF model in the N-Triples format and a mapping in the form of a rule file. These rules are then applied to the input triples and the resulting RDF graph can be serialized to RDF/XML, N3, or any other format supported by the Jena Semantic Web Framework [Jen], which is used to manipulate the input. The interesting part for us in this process is the structure of the rule file. It is represented in an RDF/XML syntax and besides the usual RDF and namespace declarations it mainly consists of a set of rules. Each rule is made up of one or more premises, one or more consequences, and optionally a series of calls to other rules with the possibility of passing values from the current triple as parameters. The premises are composed of an RDF triple that will be matched against the input triples. The values of the subject, predicate, and object are defined either as URIs of resources or as regular expressions and in the case of objects as text of literals. The consequences are composed of an RDF triple as well, but this triple provides a template for the translated triple. Its subject, predicate, and object can contain URIs, parameter variables, or one of a number of built-in functions as well as text for the object values. To reduce the effort of defining rules and to enhance the flexibility of the transformation, it is possible to call a sequence of other rules from within one rule and pass them values from the current triple. Listing 2.2 shows an example for such a rule.

```
<rdft:Rule rdf:about="sample-rules#r1">
  <rdft:order>0</rdft:order>
  <rdft:premise>
    <rdft:Premise rdf:about="sample-rules#r1_premise0">
      <rdft:order>0</rdft:order>
```

```
      <rdft:subject></rdft:subject>
      <rdft:predicate>rdf:type</rdft:predicate>
      <rdft:object>http://www.marcont.org/ont/Material</rdft:object>
    </rdft:Premise>
  </rdft:premise>
  <rdft:consequent>
    <rdft:Consequent rdf:about="sample-rules#r1_consequent0">
      <rdft:order>0</rdft:order>
      <rdft:subject>{marcont:clone($PS0, 'marcont:')}</rdft:subject>
      <rdft:predicate>rdf:type</rdft:predicate>
      <rdft:object>marcont:Book</rdft:object>
    </rdft:Consequent>
  </rdft:consequent>
</rdft:Rule>
```

Listing 2.2: RDF Translator sample rule [Kru]

These rule files must be defined by hand for every desired mapping, but the translation happens automatically without further user interaction.

Chapter 3

Requirements for an Ontology Mapping Language

Ontology designers employ different approaches to represent similar concepts in their ontologies. While some use relatively flat structures others build more complex and nested hierarchies. For the definition of our mapping language we can differentiate between two general cases. In the first case, we have the similar element represented as a single and independent property in both the source and in the target ontology. We call this a simple mapping and it can be handled easily by a one to one mapping. The second case is more complex, because the similar element is nested in a substructure in at least one of the involved ontologies. If so, to create a mapping we have to be able to extract individual elements from existing substructures in the source data or construct a fitting one in the target ontology. In the worst case, if there are substructures on both ends, we even need to perform both steps and create a new substructure from an existing, but different one. In addition to this general cases that can be expected in a lot of ontologies, other special cases can turn up that should be handled in a mapping language as well.

This section explores multiple ontologies from two domains to find the differences in the representation of similar concepts. With this knowledge gained, we can derive the requirements for our ontology mapping language.

3.1 Contact Data

We first take a look at the domain of contact data. Thereby, we present three different ontologies that, while not covering completely the same aspects, overlap sufficient in this area. One is the 'Friend of a Friend' (FOAF) project [BM07], which is a collaborative effort among Semantic Web developers to describe people and their relationships. Although it is still a work in progress with only parts of the vocabulary considered stable, it is in widely use on the Semantic Web. Another one is the vCard ontology which comes in two very different versions. Both attempt to represent the vCard Internet standard (defined by the IETF in RFC 2426 [DH98]) in an RDF encoding. Because of their many differences they are each addressed in their own section. At last is the 'Semantic Web for Research Communities' (SWRC) ontology, that describes knowledge about researchers and research communities [SBH+05].

In the following subsections we shortly introduce every ontology, then point out its specialties and analyze their impact on our mapping language.

3.1.1 FOAF

The FOAF ontology focuses on the description of people since they are mostly the link between other kinds of things we describe on the Web. For instance: people create documents, attend meetings, and are depicted in pictures. Therefore, FOAF provides at its core the class *Person* with a set of properties to describe the particular person and to link it with other things as well as other persons. The outcomes of this are not only descriptions of single persons but a linked information system, like the Web itself. The details of the FOAF ontology are defined in the FOAF Vocabulary Specification [BM07].

FOAF defines a relatively flat hierarchy with four top level classes and eight subclasses that appear as domain restrictions of most FOAF properties. From the over fifty properties defined by FOAF, currently only a handful are marked stable and are in wide use. They can be coarsely segmented in two groups, with one being simple properties that either take literals or arbitrary resources as values and therefore are candidates for simple mappings as described in the introduction of this chapter. An example for such a property is *firstName* that takes a simple literal as value. Another noteworthy fact about literals in FOAF is that they are all untyped, for instance the value of the *birthday* property has not the *date* datatype from XML Schema, as we might expect. This may cause problems if we try to map data from FOAF to an ontology with typed literals and the string representation in FOAF does not match the representation of the target datatype. The other group consists of properties with FOAF classes as values. If these value classes are blank nodes with the only purpose of connecting the primary class with additional metadata, this creates a substructure situation that needs a more complex mapping to extract this nested data.

3.1.2 vCard (2001)

The first attempt to define an ontology for the representation of vCard objects started in 2001. Its intention was not to create a separate definition of the vCard schema, but solely a RDF/XML encoding for it. The RDF vCard ontology stays close to the original vCard specification and does not introduce any capabilities not expressible in the primary format. It uses special XML and RDF functionality only to better articulate the original intentions of the vCard authors, for instance instead of the vCard type *VERSION* it uses XML Namespaces to record version information [Ian01].

RDF vCard defines no classes, but rather all metadata is expressed by properties that are connected with an URI identifying the vCard of a person. The properties have the same names as the vCard types from the original specification. The majority of them have strings as their values, except a number of special cases:

Multiple values If an RDF property has multiple values it is not just repeated like in other ontologies, instead all values are packed into an RDF Container (that is a Bag, a Sequence or an Alternative, depending on the context) and this container node acts as the value of the property. This means that the value of a property can either be a literal, if only one such element exists, or it can be a resource (the container holding the multiple string values).

Grouping In the original vCard specification it is possible to group arbitrary elements with the vCard type GROUP. In the RDF vCard ontology exists an equivalent GROUP property with an RDF Bag as the value. Every entry of the Bag is a blank node connecting it with the elements forming the group and thus creating deeply nested substructures

Type parameters A number of vCard properties include the ability to indicate one or more so-called type parameters of a value. For instance, to indicate that a certain telephone number is a fax number we can save this information along with the value. In RDF vCard this is done in two separate ways depending of the origin of the type parameters. If they are defined

by the vCard specification, it is done with an *rdf:value* property for the actual value and a number of *rdf:type* properties for the type parameters. These properties are connected by a blank node to the top level property (*TEL* in the example). If the parameters are defined by an external body, only one type parameter is allowed and it is written as an XML attribute. Therefore, we face the problem that one and the same property can have three kinds of values: just a literal if no type parameters are given, a blank node with value and type properties, or a literal as value and a type parameter as attribute to the original property.

Structured properties Three of the vCard properties are defined as structured properties. For example the property *N* represents a name structure and is composed of simple properties for the family name (*Family*), the first name (*Given*), and so on. This structure is represented in RDF as a blank node that acts as a connection element between the top level property and the secondary properties. The other two such properties are *ADR* for postal addresses and *ORG* for information about organizations.

Inline binary values A number of vCard properties like *PHOTO* allow either to reference an external resource using an URI or to store the binary resource in encoded form as a text string. In this inline form, the type of encoding is captured as an XML attribute in the PHOTO element. This leads to the problem that the value of such a property can either be a resource or a literal.

In summary, we can clearly see the lack of best practices in RDF ontology design. Although most of the properties have simply literals as values and are not nested in any way, the special cases can cause greater trouble or even impede the mapping of certain information.

3.1.3 vCard (2006)

In 2001, as the first RDF vCard ontology was defined, RDF was a novel technology and a work in progress. Best practices in designing RDF ontology had not yet evolved, which resulted in the not so easy to process ontology described in Section 3.1.2. Over the years, this changed as RDF became a W3C Recommendation in 2004 and RDF best practices emerged through the experiences made while working with it. Therefore in 2006, Norman Walsh made another attempt to define a vCard ontology in RDF [Wal05b] [Wal05a] that was later picked up by the W3C for further development [HSW06]. Like the standard from 2001, the version from 2006 tries to model a subset of the original vCard standard with the goal of representing existing vCard data in RDF.

Due to the use of modern RDF best practices, this vCard ontology is much simpler to process than the 2001 version. It consists of only five classes and about fifty properties. The main class is called *VCard* and it acts as the subject for most properties. The other classes are only defined to hold the structured properties known from the vCard specification, like the *Name* class which represents the *N* type in vCard. With the exception of this structured properties, all other properties are flat either taking a literal or the URI of any resource as value. Adhering to the RDF best practices, none of the special cases from the 2001 version exist anymore.

3.1.4 SWRC

The SWRC ontology [SBH+05] aims to do the same for researchers as FOAF does for general people, of course with a stronger focus on research related concepts. SWRC models knowledge about researchers, research communities, their publications, and activities as well as their mutual interrelations. The ontology consists of the six top level classes *Person*, *Publication*, *Event*, *Organization*, *Topic*, and *Project* each with several specialization in totally over 40 subclasses. For our purposes

here, the *Person* class is the most important. It is one of the most important in the ontology too, which is why this class appears in a large number of domain or range restriction of properties.

The ontology makes plenty use of inheritance with the definition of specialized subclasses and subproperties. This causes no problems, because these subproperties just lead to multiple individual mappings. As SWRC uses classes to encapsulate concepts as well, we receive similar substructures like in FOAF and can therefore address those the same way. Another specialty are properties like *homepage* and *email* that have string values, whereas other ontologies use resources for equal concepts.

3.2 Event Data

In this section, we present ontologies that are suitable to describe events. RDF Calendar is the first one and it is intended for recording data about events, especially calendar events. While the scope of the other two ontologies is broader, SWRC and the Semantic MediaWiki contain elements to depict event data as well.

In the remainder of this section, we first introduce briefly each of the three ontologies and then try to find its particular specialties that could have an impact on our ontology mapping language.

3.2.1 RDF Calendar

RDF Calendar [CM05] is not a new format of the Semantic Web, but it is an attempt to create an RDF representation of the popular iCalendar format. iCalendar is an industry standard created by the Network Working Group of the Internet Engineering Task Force (IETF) and it is defined in the RFC 2445 [DS98]. The aim of this project was to define a standard for representing and exchanging calendaring and scheduling data. Today, the standard is mature and widely used in many applications, so there is a lot of data available that could be useful in the Semantic Web. Due to the size of the primary standard, the RDF ontology was not defined by hand, but generated through an automated mapping approach. Unfortunately, this resulted in an ontology inconsistent with current RDF best practices.

The majority of properties take literals as values, from which only a part are associated with a datatype. There are also properties with the range restricted to specific classes. These classes are defined within the same ontology but act only as types for blank nodes that link the initial property with further properties. This yields again to data nested in substructures. An example for this kind of classes is the *attendee* property which has a range restriction of *Value_CAL-ADDRESS* class. In example data provided by the creators, the value of this property is always a blank node with more properties grouped together. Our inspection of the same test data revealed another detail in the use of this ontology. Several properties are defined as untyped literals, but they use varying datatypes in real world data. Additionally, RDF Calendar offers two types to express the time span of an event. We can either use the start (*dtstart*) and end time (*dtend*) or the start time and duration (*duration*). This flexibility cannot be expected from all ontologies, so this creates the need to convert between these representations to restore this implicit knowledge in another ontology.

3.2.2 SWRC

The SWRC ontology as introduces in Section 3.1.4 provides us also with a class *Event* and a number of corresponding properties. These are build and used in the same way as the *Person* class and its properties. Therefore, the same remarks and conclusions apply.

3.2.3 Semantic MediaWiki

The Semantic MediaWiki (SMW) is an extension of the MediaWiki[1], a widely used wiki-engine that also powers Wikipedia[2] [KVV06]. The goal of SMW is the enhancement of current wikis with semantic metadata to make their content machine-readable. These extensions need to be syntactical close to the current elements to accomplish an easy adoption by the users and a gentle transformation to the Semantic Wiki. For this purpose, SMW employs semantic annotations that can be viewed as an extension of the existing system of categories in MediaWiki. Categories are used to classify articles according to certain criteria. SMW extends this mechanism with properties that enable users to describe the semantics of any link or text on a page in machine-readable form. This metadata is used by the Semantic MediaWiki itself (for instance in more powerful search functions) or it can be exported as RDF for processing by external tools [SMW08].

The original MediaWiki has no predefined set of categories, but rather every user can create his own category by using special markup. Likewise, the Semantic MediaWiki has no given set of vocabulary terms and users are free to define their own. With the definition of every category or property a page is created where a description of the term and its use can be placed. This facilitates reuse and the correct understanding of these terms. Due to the fact that only text or links can be described with this mechanism, it becomes clear that the resulting RDF graph consists solely of simple triples with either a literal or an URI as value. This URI refers either to another page in the wiki or an external resource and the literals can be typed as well as untyped [SMW08].

The first real world application of the Semantic MediaWiki is Ontoworld.org[3] which promotes itself as the wiki for the Semantic Web community. Therefore, one of its main features is the tracking of events on Semantic Web topics. This description of event data is the reason why we take a look at it in this section. Two examples of that are the widely used properties *Start_date* and *End_date* to describe the time span of an event. The values of both are typed literals with the XML datatype *dateTime*. Because of the flexible sort of property definition in the Semantic MediaWiki, it is also possible that other users would represent a time span as a pair of start date and duration properties, which induces the need for a conversion between these two types of representation for the same information.

3.3 Summary of Requirements and Examples

In this section, we summarize the requirements found in the analysis of the different ontologies in this chapter. We first describe each requirement briefly and then illustrate it with an example.

3.3.1 Simple One to One Mapping

As expected, many mappings in various ontologies will be straightforward replacements of the property terms. These properties stand independent in the source and the target ontologies with values of literals or simple URIs to resources. Let us illustrate this situation in an example. Listing 3.1 shows the base data of our example representing a simple vCard of the person identified by the URI *ex:Bob*. It contains two triples of which both feature *ex:Bob* as subject. The first triple with the property *rdf:type* only indicates that this data is a vCard. The second and for our case more interesting triple is the one with the *vCard:fn* property. It represents one of the simple properties that are candidates for one to one mappings, because it is not nested in a substructure in the source data. To fully qualify for a simple one to one mapping, the concept of the property must

[1] http://www.mediawiki.org
[2] http://www.wikipedia.org
[3] http://ontoworld.org

further be represented in a similar unnested way in the target ontology, which is given in the case of FOAF with the corresponding property *foaf:name*.

```
@prefix ex:    <http://www.example.net/persons#> .
@prefix rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix vCard: <http://www.w3.org/2006/vcard/ns#> .

ex:Bob rdf:type vCard:VCard ;
       vCard:fn "Bob Miller" .
```

Listing 3.1: Example data for simple one to one mappings

The transformation from vCard to FOAF can be easily achieved with the simple SPARQL CONSTRUCT query of Listing 3.2. It mainly extracts the value from the *vCard:fn* property and puts it as the value of the *foaf:name* property. Additionally, the query changes the *rdf:type* property to the corresponding value from the FOAF ontology.

```
PREFIX rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf:  <http://xmlns.com/foaf/0.1/>
PREFIX vCard: <http://www.w3.org/2006/vcard/ns#>

CONSTRUCT {
  ?x rdf:type  foaf:Person ;
     foaf:name ?name .
}
WHERE {
  ?x rdf:type vCard:VCard ;
     vCard:FN ?name .
}
```

Listing 3.2: Example SPARQL query for simple one to one mappings

After the execution of the query the resulting document looks like Listing 3.3.

```
@prefix ex:   <http://www.example.net/persons#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

ex:Bob rdf:type  foaf:Person ;
       foaf:name "Bob Miller" .
```

Listing 3.3: Example results for simple one to one mappings

3.3.2 Untyped to Typed Mapping

In this case, we have an untyped literal in the source ontology but a typed one in the target ontology. Ideally, a mechanism would be given to convert the source value into the expected form of the target and type it with the correct datatype. As an example, to illustrate this kind of mapping we use the FOAF file in Listing 3.4 that uses the property *foaf:based_near*. It takes a geographical position in the form of a latitude and a longitude from the Basic Geo Vocabulary

[Bri06] as value. Even though the values of *geo:lat* and *geo:long* are represented as doubles, it is not explicitly stated in the ontology so that they are treated as strings.

```
@prefix ex:   <http://www.example.net/persons#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix geo:  <http://www.w3.org/2003/01/geo/wgs84_pos#> .
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

ex:Bob rdf:type        foaf:Person ;
       foaf:based_near [
         rdf:type      geo:Point ;
         geo:lat       "47.414524" ;
         geo:long      "8.549832"
       ] .
```

Listing 3.4: Example data for untyped to typed mappings

SPARQL does not provide a built-in function to add datatypes to untyped literals. It is not possible to use *rdf:type*, because this property can only be used with resources and it states that the subject of the respective triple is an instance of the class appearing as object. Therefore, we need to create extensions to SPARQL in the form of property functions to obtain this power. Such functions can be used to extend the functionality of SPARQL in a flexible way. Their biggest drawback is the loss of interoperability with other SPARQL engines. Normal SPARQL queries, that use only standard features, can be used with different SPARQL engines, whereas extension functions are tied to the target engine. In these examples and the rest of this work, we use the SPARQL engine ARQ[4] which is part of the Jena Semantic Web Framework[5]. Instead of defining different property functions for every possible datatype conversion, we define three generic property functions to cover this requirement. The first is called *datatype* and is used to denote the kind of datatype to add. This property takes an URI of a datatype as object, which is then stored in the execution context of ARQ, the SPARQL engine. The corresponding Java source code is depicted in Listing 3.5. If we use this kind of properties to form triples, we create not normal, but virtual triples. Such triples are not matched against the underlying ontology graph, they are executed as functions and can thereby modify values and bind variables even with values not present in the graph [KBS07].

```
package ch.uzh.ifi.rdftransformer.sparqlext;

import com.hp.hpl.jena.graph.Node;
import com.hp.hpl.jena.sparql.engine.ExecutionContext;
import com.hp.hpl.jena.sparql.engine.QueryIterator;
import com.hp.hpl.jena.sparql.engine.binding.Binding;
import com.hp.hpl.jena.sparql.pfunction.PFLib;
import com.hp.hpl.jena.sparql.pfunction.PFuncSimple;
import com.hp.hpl.jena.sparql.util.Symbol;

/**
 * This class is an ARQ property function. It stores the name of the
 * datatype in the ARQ execution context. It must be executed prior to
```

[4] http://jena.sourceforge.net/ARQ/
[5] http://jena.sourceforge.net

```
 * a call to the addDatatype property function to set the datatype.
 * @author Matthias Hert
 *
 */
public class datatype extends PFuncSimple {

    @Override
    public QueryIterator execEvaluated(Binding binding, Node subject,
            Node predicate, Node object, ExecutionContext execCxt) {
        execCxt.getContext().set(Symbol.create("datatype"),
                object.toString(false));

        return PFLib.result(binding, execCxt);
    }
}
```

Listing 3.5: Java source code for property function *datatype*

The second property function is named *datatypeConverter* and is optionally used to specify a converter class that translates the syntax representation of the input data into the format of the desired output. It is implemented like the *datatype* property function as it stores the fully qualified name of a Java class in the ARQ execution context. Its source code is depicted in Appendix A.1. In our example, both ontologies use doubles, so we do not need this, but other circumstances are possible where this functionality becomes beneficial. *addDatatype* is the third property function and performs the actual creation of the typed literal from the untyped one. Before we can invoke this function, we first need to set the desired datatype with the *datatype* property discussed before and optionally name a Java class for syntax conversions. After that, the untyped literal is passed as the subject and serves as value for the new typed literal. The *addDatatype* function first reads the datatype and converter class names from the ARQ execution context. If a converter class is set, it tries to instantiate the class and performs the transformation, else it leaves the input untouched. In the end, it uses that value to create a new typed literal with a possible language tag carried over without change and assigns it to the object of this triple for further use in the SPARQL query. Listing 3.6 depicts the Java source code of this property function.

```
package ch.uzh.ifi.rdftransformer.sparqlext;

import org.apache.log4j.Logger;

import ch.uzh.ifi.rdftransformer.sparqlext.IDatatypeConverter;

import com.hp.hpl.jena.datatypes.BaseDatatype;
import com.hp.hpl.jena.datatypes.RDFDatatype;
import com.hp.hpl.jena.datatypes.TypeMapper;
import com.hp.hpl.jena.graph.Node;
import com.hp.hpl.jena.sparql.core.Var;
import com.hp.hpl.jena.sparql.engine.ExecutionContext;
import com.hp.hpl.jena.sparql.engine.QueryIterator;
import com.hp.hpl.jena.sparql.engine.binding.Binding;
```

```
import com.hp.hpl.jena.sparql.pfunction.PFLib;
import com.hp.hpl.jena.sparql.pfunction.PFuncSimple;
import com.hp.hpl.jena.sparql.util.Symbol;

/**
 * This class is an ARQ property function. It is used to add a
 * datatype to an untyped literal and optionally to convert the
 * syntax of the input.
 * @author Matthias Hert
 *
 */
public class addDatatype extends PFuncSimple {

    private static final Logger log = Logger
            .getLogger(addDatatype.class);

    @Override
    public QueryIterator execEvaluated(Binding binding, Node subject,
            Node predicate, Node object, ExecutionContext execCxt) {
        // return unchanged if subject is no literal
        if (!subject.isLiteral()) {
            return PFLib.oneResult(binding, Var.alloc(object),
                    subject, execCxt);
        }

        // get name of datatype
        String datatype = execCxt.getContext().getAsString(Symbol
                .create("datatype"));
        log.debug("datatype: " + datatype);
        // get name of optional datatype converter class
        String converter = execCxt.getContext().getAsString(Symbol
                .create("datatypeConverter"));
        log.debug("datatypeConverter: " + converter);
        // remove entries from execution context for further calls
        execCxt.getContext().remove(Symbol.create("datatype"));
        execCxt.getContext().remove(
                Symbol.create("datatypeConverter"));

        RDFDatatype type = null;
        String resultValue = null;

        if (datatype != null) {
            // handle XML Schema datatypes
            if (datatype.startsWith(
                    "http://www.w3.org/2001/XMLSchema#")) {
```

```
                type = TypeMapper.getInstance()
                        .getTypeByName(datatype);
            }
            // handle all other datatypes
            else {
                type = new BaseDatatype(datatype);
            }
        }

        // perform syntax conversion if requested
        if (converter != null) {
            IDatatypeConverter conv = null;
            try {
                conv = (IDatatypeConverter)Class.forName(converter)
                    .newInstance();
            }
            catch (IllegalAccessException ex) {
                throw new ClassLoadingException(
                        "Could not access converter class!");
            }
            catch (InstantiationException ex) {
                throw new ClassLoadingException(
                        "Could not instantiate converter class!");
            }
            catch (ClassNotFoundException ex) {
                throw new ClassLoadingException(
                        "Converter class not found!");
            }
            resultValue = conv.convert(
                    subject.getLiteralLexicalForm());
        }
        // apply without conversion
        else {
            resultValue = subject.getLiteralLexicalForm();
        }

        // create typed result
        Node result = Node.createLiteral(resultValue,
                subject.getLiteralLanguage(), type);
        return PFLib.oneResult(binding, Var.alloc(object),
                result, execCxt);
    }
}
```

Listing 3.6: Java source code for property function *addDatatype*

In Listing 3.7 we can see how this self-defined property function is used. In order that we can access this function, we have to assign it with a namespace. This is done with the pseudo URI scheme *java* that looks for the code on the Java classpath and loads it dynamically. After this step, we can use a property function like any other property.

```
PREFIX rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX geo:   <http://www.w3.org/2003/01/geo/wgs84_pos#>
PREFIX foaf:  <http://xmlns.com/foaf/0.1/>
PREFIX fn:    <java:ch.uzh.ifi.rdftransformer.sparqlext.>
PREFIX vCard: <http://www.w3.org/2006/vcard/ns#>
PREFIX xsd:   <http://www.w3.org/2001/XMLSchema#>

CONSTRUCT {
  ?x  rdf:type        vCard:VCard ;
      vCard:geo       _:l .
  _:l rdf:type        vCard:Location ;
      vCard:latitude  ?latitude ;
      vCard:longitude ?longitude .
}
WHERE {
  ?x    rdf:type         foaf:Person ;
        foaf:based_near  ?p .
  ?p    geo:lat          ?lat ;
        geo:long         ?long .
  ?lat  fn:datatype      xsd:double ;
        fn:addDatatype   ?latitude .
  ?long fn:datatype      xsd:double ;
        fn:addDatatype   ?longitude .
}
```

Listing 3.7: Example SPARQL query for untyped to typed mappings

With the help of our property functions, a SPARQL engine can transform the untyped literals of the geo properties into the typed ones of the vCard ontology. The resulting document is shown in Listing 3.8.

```
@prefix ex:    <http://www.example.net/persons#> .
@prefix rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix vCard: <http://www.w3.org/2006/vcard/ns#> .
@prefix xsd:   <http://www.w3.org/2001/XMLSchema#> .

ex:Bob rdf:type          vCard:VCard ;
       vCard:geo         [
         rdf:type        vCard:Location ;
         vCard:latitude  "47.414524"^^xsd:double ;
         vCard:longitude "8.549832"^^xsd:double
       ] .
```

Listing 3.8: Example results for untyped to typed mappings

3.3.3 Extracting Nested Data

The source ontology may contain data in nested structures, but the target data represents the same concepts with independent properties. Therefore, we need the ability to extract this data from its nesting. The transformation from the *vCard:n* data structure to the corresponding properties in FOAF is an example for a mapping of this kind. Listing 3.9 shows the data for this example with the *vCard:n* property that takes a blank node of the type *vCard:Name* as value. This blank node in turn consists of nested properties like *vCard:family-name* and *vCard:first-name* as used in the example data.

```
@prefix ex:    <http://www.example.net/persons#> .
@prefix rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix vCard: <http://www.w3.org/2006/vcard/ns#> .

ex:Bob rdf:type            vCard:VCard ;
       vCard:n             [
         rdf:type          vCard:Name ;
         vCard:family-name "Miller" ;
         vCard:first-name  "Bob"
       ] .
```

Listing 3.9: Example data for extracting nested data

To translate this data into another ontology, we have to break up the nesting. In our example, the target is the FOAF ontology where the first and family name concepts are top level properties. Listing 3.10 depicts a SPARQL query that performs this translation for our example data. It uses the variable *?n* in the WHERE clause to bind the blank node and gets thereby to the real data, which it assigns then to the respective FOAF properties in the CONSTRUCT clause of the query.

```
PREFIX rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf:  <http://xmlns.com/foaf/0.1/>
PREFIX vCard: <http://www.w3.org/2006/vcard/ns#>

CONSTRUCT {
  ?x rdf:type         foaf:Person ;
     foaf:family_name ?familyName ;
     foaf:firstName   ?firstName .
}
WHERE {
  ?x rdf:type          vCard:VCard ;
     vCard:n           ?n .
  ?n vCard:family-name ?familyName ;
     vCard:first-name  ?firstName .
}
```

Listing 3.10: Example SPARQL query for extracting nested data

The results of this transformation process are shown in Listing 3.11. From the source data, only the values of the first and family names are preserved, the blank node is no longer needed and thus eliminated.

```
@prefix ex:   <http://www.example.net/persons#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

ex:Bob rdf:type         foaf:Person ;
       foaf:family_name "Miller" ;
       foaf:firstName   "Bob" .
```

Listing 3.11: Example results for extracting nested data

3.3.4 Create Substructures

This is the reverse of the prior case. The properties in the source data consist of independent properties, but the target ontology expects a structured representation. Therefore, we need a way to create such structures. For illustration, we can take the same example as in the previous case, this time only in the reverse direction. We start with the FOAF data in Listing 3.11 and transform it back in the form shown in Listing 3.9. To achieve this, we use the SPARQL query depicted in Listing 3.12. In its CONSTRUCT clause it creates a new blank node of the type *vCard:Name* and then attaches the *vCard:family-name* and *vCard:first-name* properties to it and fills them with the values from the respective FOAF properties.

```
PREFIX rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf:  <http://xmlns.com/foaf/0.1/>
PREFIX vCard: <http://www.w3.org/2006/vcard/ns#>

CONSTRUCT {
  ?x  rdf:type          vCard:VCard ;
      vCard:n           _:n .
  _:n rdf:type          vCard:Name ;
      vCard:family-name ?familyName ;
      vCard:first-name  ?firstName .
}
WHERE {
  ?x rdf:type         foaf:Person ;
     foaf:family_name ?familyName ;
     foaf:firstName   ?firstName .
}
```

Listing 3.12: Example SPARQL query for creating substructures

The execution of the query on the example data reveals that the results are indeed the same as in Listing 3.9.

3.3.5 Converting Structures

It is also possible that the last two points appear together, that means the data in the source ontology is packed into a substructure and must be converted into a different one in the target vocabulary. Ideally, this should not require additional constructs, only the combination of the solutions

to the last two problems. To illustrate this case, we use again the example data from Listing 3.9 containing the property *vCard:n* with the nested properties *vCard:family-name* and *vCard:first-name* as value. We use a fictional ontology as target that basically has the same structure, but with different named properties from another namespace. Listing 3.13 shows the SPARQL query that transforms our example source data into the target ontology. For that purpose, it extracts the values of the nested properties from the source data, creates the necessary structures of the target ontology in the same way as described in the prior case, and then fills it with the corresponding data values.

```
PREFIX exont: <http://www.example.net/ontology#>
PREFIX rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX vCard: <http://www.w3.org/2006/vcard/ns#>

CONSTRUCT {
  ?x     rdf:type         exont:Person ;
         exont:name       _:name .
  _:name rdf:type         exont:Name ;
         exont:familyName ?familyName ;
         exont:firstName  ?firstName .
}
WHERE {
  ?x rdf:type          vCard:VCard ;
     vCard:n           ?n .
  ?n vCard:family-name ?familyName ;
     vCard:first-name  ?firstName .
}
```

Listing 3.13: Example SPARQL query for converting structures

In Listing 3.14 we see the result of the translation with the new structured property from the target ontology and the respective values in its nested properties.

```
@prefix ex:    <http://www.example.net/persons#> .
@prefix exont: <http://www.example.net/ontology#> .
@prefix rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

ex:Bob rdf:type           exont:Person ;
       exont:name         [
         rdf:type         exont:Name ;
         exont:familyName "Miller" ;
         exont:firstName  "Bob"
       ] .
```

Listing 3.14: Example results for converting structures

3.3.6 Literals to URIs

Ontologies may use plain strings for things like email addresses or website URLs, whereas others represent those concepts as URIs. A mapping language should offer a possibility to convert such

literals to real URIs and vice versa. In our illustrative example, we use the SWRC ontology in the source data. It uses plain strings to represent email addresses whereas FOAF, our target ontology for this example, uses URIs. Listing 3.15 shows the sample RDF document with the *swrc:email* property.

```
@prefix ex:   <http://www.example.net/persons#> .
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix swrc: <http://swrc.ontoware.org/ontology#> .

ex:Bob rdf:type   swrc:Person ;
       swrc:email "bob@example.net" .
```

Listing 3.15: Example data for literals to URIs mappings

In order to convert the literal value into an URI, we first have to extract the string value from the source data and prefix it according to the *mailto* URI scheme. We then have to turn the URI string into a resource. This is not possible with normal SPARQL operations, but again it can be achieved with property functions. Email addresses are not the only example for this kind of conversions, in fact everything represented as URI can as well be described as plain strings in another ontology. Therefore, our approach to this problem is generic and extendable. We define the property function *convertURI* that performs the actual mapping but delegates the creation of the URI string or the literal in each case to a separate class. Such a class can be written for the translation of every URI scheme, like *mailto* in our example. The only precondition for such classes is that they implement the Java interface *IURIConverter* that contains only two methods: *toURI* for converting literals to URIs and *toLiteral* for converting URIs back to literals. In Listing 3.16 we see such a class for the URI scheme *mailto*. It extends the Java class *URIConverterBase* that is a convenience class with default implementations of the two converter methods.

```
package ch.uzh.ifi.rdftransformer.sparqlext;


/**
 * Converter class for the mailto URI scheme.
 * @author Matthias Hert
 *
 */
public class mailto extends URIConverterBase {

    /**
     * Converts a literal to a mailto URI. Adds the prefix
     * 'mailto:' if not already present.
     * @param literal The email address in literal form
     * @return The mailto URI String
     */
    @Override
    public String toURI(String literal) {
        if (literal.startsWith("mailto:")) {
            return literal;
        }
```

```
        else {
            return "mailto:" + literal;
        }
    }

    /**
     * Converts a mailto URI to a literal. Strips the prefix
     * 'mailto:' if present.
     * @param uri The email address in URI form
     * @return The email address as String
     */
    @Override
    public String toLiteral(String uri) {
        if (uri.startsWith("mailto:")) {
            return uri.substring("mailto:".length());
        }
        else {
            return uri;
        }
    }
}
```

Listing 3.16: Java source code of *mailto* class

We can configure which converter class will be used with the property function *uriConverter*. It takes a string with the fully qualified name of a Java class as object and stores this value in the ARQ execution context, where it is retrieved by the *convertURI* class to select the right conversion. Obviously, this property function has to be called before the execution of the *convertURI* function. Listing 3.17 shows the Java source code of the *uriConverter* property function.

```
package ch.uzh.ifi.rdftransformer.sparqlext;

import com.hp.hpl.jena.graph.Node;
import com.hp.hpl.jena.sparql.engine.ExecutionContext;
import com.hp.hpl.jena.sparql.engine.QueryIterator;
import com.hp.hpl.jena.sparql.engine.binding.Binding;
import com.hp.hpl.jena.sparql.pfunction.PFLib;
import com.hp.hpl.jena.sparql.pfunction.PFuncSimple;
import com.hp.hpl.jena.sparql.util.Symbol;

/**
 * This class is an ARQ property function. It stores the fully
 * qualified name of the Java converter class used to convert
 * between literals and URIs for a certain URI scheme in the
 * ARQ execution context.
 * @author Matthias Hert
 *
```

```
 */
public class uriConverter extends PFuncSimple {

    @Override
    public QueryIterator execEvaluated(Binding binding, Node subject,
            Node predicate, Node object, ExecutionContext execCxt) {
        execCxt.getContext().set(Symbol.create("uriConverter"),
                object.toString(false));

        return PFLib.result(binding, execCxt);
    }
}
```

Listing 3.17: Source code of the Java class for the *uriConverter* property function

As already mentioned, the *convertURI* class performs the actual conversion. Listing 3.18 depicts the source code of this class. We can see that it first retrieves the name of the converter class from the execution context and then tries to instantiate that class. If this succeeds, it uses this class to convert the subject either to an URI resource or to a literal depending on the type of the subject. The result is then assigned to the object of the triple where this property function is used.

```
package ch.uzh.ifi.rdftransformer.sparqlext;

import com.hp.hpl.jena.graph.Node;
import com.hp.hpl.jena.sparql.core.Var;
import com.hp.hpl.jena.sparql.engine.ExecutionContext;
import com.hp.hpl.jena.sparql.engine.QueryIterator;
import com.hp.hpl.jena.sparql.engine.binding.Binding;
import com.hp.hpl.jena.sparql.pfunction.PFLib;
import com.hp.hpl.jena.sparql.pfunction.PFuncSimple;
import com.hp.hpl.jena.sparql.util.Symbol;

/**
 * This class is an ARQ property function. It performs the
 * transformation from literals to URIs or from URIs to
 * literals depending on the input subject. It uses an URI
 * scheme specific converter class for the syntax conversion
 * of the input.
 * @author Matthias Hert
 *
 */
public class convertURI extends PFuncSimple {

    @Override
    public QueryIterator execEvaluated(Binding binding, Node subject,
            Node predicate, Node object, ExecutionContext execCxt) {
        // get name of converter class
```

```
        String converter = execCxt.getContext().getAsString(Symbol
                .create("uriConverter"));
        // remove entry from execution context for further calls
        execCxt.getContext().remove(Symbol.create("uriConverter"));

        String literal = null;
        String result = null;
        IURIConverter conv = null;

        // create an instance of the converter class
        try {
            conv = (IURIConverter)Class.forName(converter)
                .newInstance();
        }
        catch (IllegalAccessException ex) {
            throw new ClassLoadingException(
                    "Could not access converter class!");
        }
        catch (InstantiationException ex) {
            throw new ClassLoadingException(
                    "Could not instantiate converter class!");
        }
        catch (ClassNotFoundException ex) {
            throw new ClassLoadingException(
                    "Converter class not found!");
        }

        // convert to URI if the subject is a literal
        if (subject.isLiteral()) {
            literal = subject.getLiteralLexicalForm();
            result = conv.toURI(literal);

            return PFLib.oneResult(binding, Var.alloc(object),
                    Node.createURI(result), execCxt);
        }
        // subject must be an URI -> convert to literal
        else {
            literal = subject.toString();
            result = conv.toLiteral(literal);

            return PFLib.oneResult(binding, Var.alloc(object),
                    Node.createLiteral(result), execCxt);
        }
    }
```

```
}
```

Listing 3.18: Java source code of property function *convertURI*

As already explained earlier in this section, we can use this property function like a normal property after the definition of the right namespace. Listing 3.19 depicts how our property function is used to transform the example SWRC data into the target FOAF ontology. Pay especially attention to the sequence of property usage. First, we set the converter class with the *uriConverter* property function and not until then we can call *convertURI* to execute the conversion.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX fn:   <java:ch.uzh.ifi.rdftransformer.sparqlext.>
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX swrc: <http://swrc.ontoware.org/ontology#>

CONSTRUCT {
  ?x rdf:type  foaf:Person ;
     foaf:mbox ?email .
}
WHERE {
  ?x rdf:type        swrc:Person ;
     swrc:email      ?m .
  ?m fn:uriConverter "ch.uzh.ifi.rdftransformer.sparqlext.mailto" ;
     fn:convertURI   ?email .
}
```

Listing 3.19: Example SPARQL query for literals to URIs mappings

The output of this query is the transformed RDF graph shown in Listing 3.20 with all email strings converted to legal URIs.

```
@prefix ex:   <http://www.example.net/persons#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

ex:Bob rdf:type  foaf:Person ;
       foaf:mbox <mailto:bob@example.net> .
```

Listing 3.20: Example results for literals to URIs mappings

3.3.7 Restoring Implicit Knowledge

Sometimes an ontology contains implicit knowledge that is expressed explicitly in another. To enable a mapping as complete as possible, we need to be able to make this implicit knowledge explicit, so that we can store it in the target vocabulary. We explain this problem on the basis of the following example using the ontology of the Semantic MediaWiki. With the SMW vocabulary we can represent the start and end date of an event as shown in Listing 3.21. It is obvious that these two properties contain the implicit knowledge of the duration of the event as well.

```
@prefix ex:  <http://www.example.net/events#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
```

```
@prefix smw: <http://smw.ontoware.org/2005/smw#> .
@prefix p: <http://ontoworld.org/wiki/Special:URIResolver/Property-3A>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .


ex:IECC2005 rdf:type     smw:Thing ;
            p:Start_date "2005-11-06T00:00:00"^^xsd:dateTime ;
            p:End_date   "2005-11-10T00:00:00"^^xsd:dateTime .
```

Listing 3.21: Example data for restoring implicit knowledge

Our fictional target ontology only features properties for the start date and the duration of an event. Therefore, we need to calculate the duration from the original start and end date properties. SPARQL offers no functionality to meet this requirement and the variants of this case are so diverse that we abandon a completely generic solution. Instead, we define two property functions *args* and *toDuration*. The first takes a list of nodes as object and stores them in the execution context of ARQ. In our example, it is used to pass the start and end date into the second function but it has applications in every other situation where property functions need an arbitrary number of arguments. Its code is shown in Listing 3.22.

```
package ch.uzh.ifi.rdftransformer.sparqlext;

import com.hp.hpl.jena.graph.Node;
import com.hp.hpl.jena.sparql.engine.ExecutionContext;
import com.hp.hpl.jena.sparql.engine.QueryIterator;
import com.hp.hpl.jena.sparql.engine.binding.Binding;
import com.hp.hpl.jena.sparql.pfunction.PFLib;
import com.hp.hpl.jena.sparql.pfunction.PFuncSimpleAndList;
import com.hp.hpl.jena.sparql.pfunction.PropFuncArg;
import com.hp.hpl.jena.sparql.util.Symbol;

/**
 * This class is an ARQ property function. It is used to store an
 * arbitrary number of arguments in the ARQ execution context. They
 * are stored in the form of a list and can be used by other property
 * functions.
 * @author Matthias Hert
 *
 */
public class args extends PFuncSimpleAndList {

    @Override
    public QueryIterator execEvaluated(Binding binding, Node subject,
            Node predicate, PropFuncArg objects,
            ExecutionContext execCxt) {
        execCxt.getContext().set(Symbol.create("args"),
                objects.getArgList());
```

```
        return PFLib.result(binding, execCxt);
    }
}
```

Listing 3.22: Java source code of property function *args*

toDuration is the second and more specialized property function for this case. Listing 3.23 depicts its source code in Java. This property function is only invoked after two arguments are set in the ARQ execution context via the *args* property function. It first loads the two parameters representing the start and end date and then calculates the duration from those values. After that, the result is used to create a new typed literal that is lastly assigned to the object of the respective triple.

```
package ch.uzh.ifi.rdftext;

import java.util.List;

import javax.xml.datatype.DatatypeConfigurationException;
import javax.xml.datatype.DatatypeFactory;
import javax.xml.datatype.Duration;
import javax.xml.datatype.XMLGregorianCalendar;

import org.apache.log4j.Logger;


import ch.uzh.ifi.rdftransformer.sparqlext.ArgumentSizeException;

import com.hp.hpl.jena.datatypes.TypeMapper;
import com.hp.hpl.jena.graph.Node;
import com.hp.hpl.jena.sparql.core.Var;
import com.hp.hpl.jena.sparql.engine.ExecutionContext;
import com.hp.hpl.jena.sparql.engine.QueryIterator;
import com.hp.hpl.jena.sparql.engine.binding.Binding;
import com.hp.hpl.jena.sparql.pfunction.PFLib;
import com.hp.hpl.jena.sparql.pfunction.PFuncSimple;
import com.hp.hpl.jena.sparql.util.Symbol;

/**
 * This class is an ARQ property function. It is used to calculate
 * the duration of an event from the given start and end times.
 * @author Matthias Hert
 *
 */
public class toDuration extends PFuncSimple {

    private static final Logger log = Logger
            .getLogger(toDuration.class);
```

```
@Override
@SuppressWarnings("unchecked")
public QueryIterator execEvaluated(Binding binding, Node subject,
        Node predicate, Node object, ExecutionContext execCxt) {
    // get arguments from execution context...
    List<Node> args = (List<Node>)execCxt.getContext().get(
            Symbol.create("args"));
    // ...and remove them for future calls
    execCxt.getContext().remove(Symbol.create("args"));
    // it needs exactly two arguments
    if (args.size() != 2) {
        throw new ArgumentSizeException(
            "Wrong number of arguments! Exactly two are needed");
    }
    else {
        Node start = args.get(0);
        Node end = args.get(1);
        String durationString = null;

        try {
            // create objects of XML datatypes
            DatatypeFactory datatypeFactory = DatatypeFactory
                .newInstance();
            XMLGregorianCalendar startCal = datatypeFactory
                .newXMLGregorianCalendar(
                start.getLiteralLexicalForm());
            XMLGregorianCalendar endCal = datatypeFactory
                .newXMLGregorianCalendar(
                end.getLiteralLexicalForm());

            // calculate duration
            long durationInMillis = endCal.toGregorianCalendar()
                .getTimeInMillis() - startCal.toGregorianCalendar()
                .getTimeInMillis();

            Duration duration = datatypeFactory.newDuration(
                    durationInMillis);
            durationString = duration.toString();
        }
        catch (DatatypeConfigurationException ex) {
            log.error(ex.getLocalizedMessage());
        }

        // create result
```

```
            Node result = Node.createLiteral(durationString,
                    start.getLiteralLanguage(),
                    TypeMapper.getInstance().getTypeByName(
                    "http://www.w3.org/2001/XMLSchema#duration"));
            return PFLib.oneResult(binding, Var.alloc(object),
                    result, execCxt);
        }
    }
}
```

Listing 3.23: Java source code of property function *toDuration*

Listing 3.24 shows how the just defined property functions are used to translate between the different representations of our example ontologies. First, we invoke the *args* property function with the variables holding the start and end date in a list as the object. Then, we invoke the *toDuration* property function with an unbound variable as object that will hold the result after the completion of the function. This variable can finally be used in the CONSTRUCT clause to build the target document.

```
PREFIX exterms: <http://www.example.net/terms#>
PREFIX fn:      <java:ch.uzh.ifi.rdftransformer.sparqlext.>
PREFIX func:    <java:ch.uzh.ifi.rdftext.>
PREFIX p:   <http://ontoworld.org/wiki/Special:URIResolver/Property-3A>
PREFIX rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX smw:     <http://smw.ontoware.org/2005/smw#>

CONSTRUCT {
  ?x rdf:type         exterms:Event ;
     exterms:start    ?start ;
     exterms:duration ?duration .
}
WHERE {
  ?x rdf:type        smw:Thing ;
     p:Start_date    ?start ;
     p:End_date      ?end ;
     fn:args         (?start ?end) ;
     func:toDuration ?duration .
}
```

Listing 3.24: Example SPARQL query for restoring implicit duration knowledge

Finally, Listing 3.25 shows the results of this query applied to our example data. It now represents the event with the start date and the duration instead of the end date.

```
@prefix ex:      <http://www.example.net/events#> .
@prefix exterms: <http://www.example.net/terms#> .
@prefix rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd:     <http://www.w3.org/2001/XMLSchema#> .
```

```
ex:IECC2005 rdf:type         exterms:Event ;
            exterms:start    "2005-11-06T00:00:00"^^xsd:dateTime ;
            exterms:duration "P4DT0H0M0.000S"^^xsd:duration .
```

Listing 3.25: Example results for restoring implicit duration knowledge

The case of restoring implicit knowledge is so manifold that we feel the need to provide another example to show this diversity. Our second example is about two fictional ontologies representing metadata of music and especially CD records. Our source ontology describes each CD detailed with its name as well as the name and play time of every track. An example document with one five track CD is shown in Listing 3.26.

```
@prefix music: <http://www.example.net/music#> .
@prefix rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

music:cd1 rdf:type       music:CD ;
          music:name     "Back To Bedlam" ;
          music:track    _:track1 ;
          music:track    _:track2 ;
          music:track    _:track3 ;
          music:track    _:track4 ;
          music:track    _:track5 .
_:track1  music:name     "High" ;
          music:playtime "4:03" .
_:track2  music:name     "You're Beautiful" ;
          music:playtime "3:36" .
_:track3  music:name     "Wisemen" ;
          music:playtime "3:46" .
_:track4  music:name     "Goodbye My Lover" ;
          music:playtime "4:23" .
_:track5  music:name     "Tears And Rain" ;
          music:playtime "4:07" .
```

Listing 3.26: Example data for restoring implicit knowledge

On the other hand, our target ontology serves only as an overlook of CD collections and therefore stores only the names of CDs and their total play times. It is evident, that a document from the first ontology contains implicitly all information needed to transform it into the second one. The total play time of a CD can be easily calculated by aggregating the individual play times of each track. Keep in mind that part of the information is lost with this transformation, that means a translation back in the source ontology is not possible. Contrary to the prior example we now have a varying number of input parameters that cannot all be captured at the same time to pass them together into a property function. Therefore, we need another means to solve this problem. After consulting the Jena developers mailing list[6], Andy Seaborne, one of the creators of SPARQL and ARQ, suggested three different possibilities to do this [SH07]. The first alternative would be to pass the complete CD structure into a property function and use the functionality of Jena to find the individual play times in the graph, sum them up, and bind the result to a variable in the query. This way, the whole aggregation is hidden in a property function which makes it important to give it a descriptive name. The second possibility would be to add a triple

[6] http://tech.groups.yahoo.com/group/jena-dev/

with the calculated total play time before the query is executed. We would have to intercept the data loading process, calculate the total time manually, and add a special triple for the total play time. The transformation query could then use this extra triple as source for the target property *totalPlaytime* and would not need to make calculations itself. This method is essentially the same as the first with the calculation work moved from query to data loading and preparation time. The disadvantage of this approach is that we would have to provide a separate extension mechanism for data loading to support different kinds of input data. In contrast, the first approach would only need the already known property functions as extensions. The third way would be to extend ARQ and implement the *SUM* aggregation function. In conjunction with the already implemented *GROUP BY* function the total playtime could be calculated in the query without the need of custom property functions. The major drawback of this approach is that the query gets more complex and we would need to extend ARQ further every time we come across another kind of aggregation, like calculating an average value. A fourth possibility would be to create a property function that takes each single play time individually and uses the execution context of ARQ as a cache for the intermediate aggregation results. Due to the variable number of elements and the way SPARQL processes queries this approach would be cumbersome to implement and use. With regard to our requirements, the first approach promises to be the easiest and most flexible of all the presented possibilities, which is why we will use it and also implemented it for our current example. Listing 3.27 depicts the code of our property function *totalPlaytime*. We first retrieve the active graph containing the CDs from the execution context. Then, we use the *com.hp.hpl.jena.graph.query.QueryHandler* class provided by Jena to query for the individual tracks. Given the structure of our input ontology, we need to find all objects of triples with a given CD as subject and the *track* property as predicate. To support multiple CDs per document we pass the URI of the current CD as a parameter into the function. The ARQ Java method *objectsFor* looks for such triples and returns the matching track objects. In a second step, we iterate over the tracks in the same manner to extract the play times and sum them up. After processing all tracks in this way, we format the result and bind it to the variable passed as object.

```
package ch.uzh.ifi.rdftext;

import com.hp.hpl.jena.graph.Graph;
import com.hp.hpl.jena.graph.Node;
import com.hp.hpl.jena.graph.query.QueryHandler;
import com.hp.hpl.jena.sparql.core.Var;
import com.hp.hpl.jena.sparql.engine.ExecutionContext;
import com.hp.hpl.jena.sparql.engine.QueryIterator;
import com.hp.hpl.jena.sparql.engine.binding.Binding;
import com.hp.hpl.jena.sparql.pfunction.PFLib;
import com.hp.hpl.jena.sparql.pfunction.PFuncSimple;
import com.hp.hpl.jena.util.iterator.ExtendedIterator;

/**
 * This class is an ARQ property function. It is an example for a
 * function used in a complex mapping. It calculates the total play
 * time of a CD from the play times of its single tracks. Every track
 * must be attached to the CD subject via a http://www.example.net/
 * music#track predicate. Each track is a blank node with a
 * http://www.example.net/music#playtime predicate whose object
```

```
 * contains the play time of this track. Those are summed to the total
 * play time of a CD.
 * @author Matthias Hert
 *
 */
public class totalPlaytime extends PFuncSimple {

    @Override
    public QueryIterator execEvaluated(Binding binding, Node subject,
            Node predicate, Node object, ExecutionContext execCxt) {
        // get RDF graph
        Graph graph = execCxt.getActiveGraph();
        // get query handler for this graph
        QueryHandler qraphHandler = graph.queryHandler();
        int totalMinutes = 0;
        int totalSeconds = 0;

        // get all tracks of this CD
        ExtendedIterator trackIterator = qraphHandler.objectsFor(
                subject,
                Node.createURI("http://www.example.net/music#track"));
        // handle each track separately
        while (trackIterator.hasNext()) {
            Node track = (Node)trackIterator.next();
            // get the play time for each track
            ExtendedIterator playtimeIterator = qraphHandler
                    .objectsFor(track, Node.createURI(
                    "http://www.example.net/music#playtime"));
            while (playtimeIterator.hasNext()) {
                Node playtimeNode = (Node)playtimeIterator.next();
                String playtime = playtimeNode.getLiteralLexicalForm();
                // parse for minutes part
                int minutes = Integer.parseInt(playtime.substring(0,
                        playtime.indexOf(":")));
                // parse for seconds part
                int seconds = Integer.parseInt(playtime.substring(
                        playtime.indexOf(":") + 1));
                totalMinutes += minutes;
                totalSeconds += seconds;
            }
        }

        // calculate total minutes
        if (totalSeconds > 59) {
            totalMinutes += totalSeconds / 60;
```

```
            totalSeconds = totalSeconds % 60;
        }

        String totalSecondsLit = null;
        // prefix single-digit seconds values with a 0
        if (totalSeconds < 10) {
            totalSecondsLit = "0" + totalSeconds;
        }
        else {
            totalSecondsLit = String.valueOf(totalSeconds);
        }

        Node result = Node.createLiteral(totalMinutes + ":"
                + totalSecondsLit);
        return PFLib.oneResult(binding, Var.alloc(object), result,
                execCxt);
    }
}
```

Listing 3.27: Java source code of property function *totalPlaytime*

Listing 3.28 shows the usage of our *totalPlaytime* property function. We pass a CD as subject and an unbound variable as the object. The function gets invoked and calculates the total play time which it stores afterwards in the passed variable. This value can then be used in the construction of the target document.

```
PREFIX cds:   <http://www.example.net/cds#>
PREFIX func:  <java:ch.uzh.ifi.rdftext.>
PREFIX music: <http://www.example.net/music#>
PREFIX rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

CONSTRUCT {
  ?x rdf:type          cds:CD ;
     cds:name          ?name ;
     cds:totalPlaytime ?tpt .
}
WHERE {
  ?x rdf:type           music:CD ;
     music:name         ?name ;
     func:totalPlaytime ?tpt .
}
```

Listing 3.28: Example SPARQL query for calculating implicit total play time

After applying the previously described query on our example data from Listing 3.26, we get the new document depicted in Listing 3.29. It is expressed in the vocabulary of the target ontology and contains the results from our custom extension in the property *totalPlaytime*.

```
@prefix cds: <http://www.example.net/cds#> .
```

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

music:cd1 rdf:type          cds:CD ;
          cds:name          "Back To Bedlam" ;
          cds:totalPlaytime "19:55" .
```

Listing 3.29: Example results for calculation of total play time

3.3.8 Substitution of Class Types

In the previous cases we only needed to replace the predicates of a triple, subject and object remained the same or in the case of objects get only converted by property functions. There is one case where not a conversion of the object, but a substitution is needed. These are class types expressed in a triple with the *rdf:type* property. If we transform, for instance, a vCard document to FOAF, the subjects of the former will have a type of *vCard:VCard*, but the subjects of the later need a type of *foaf:Person*. Therefore, we need the ability to replace the objects of triples that have *rdf:type* as its predicate. This simple replacement case may remind the reader of the simple one to one mapping case described in Section 3.3.1 that also simply replaces one part of the triple. The difference lies in the replaced part. In the simple one to one mapping, we always replace the predicate of a triple, but in this case the substitution applies always to the object of a triple. Therefore, another feature is needed to address this case. For an example of this, take another look at the previous examples in this section, although never explicitly stated, we used the substitution of class types in all of those examples.

Chapter 4

Our Approach for an Ontology Mapping Language

In this chapter, we present our approach for an ontology mapping language. We describe the syntax of the mapping and show how the individual cases from Section 3.3 are handled. For every requirement we give an example of how our mapping is translated into a SPARQL query that performs the forward transformation and likewise describe the generation of the SPARQL query used for the backward transformation. The chapter is structured as follows: in the first section we generally introduce the mapping format, then we take a look at the way our mapping language uses *XML Namespaces* in the second section. The third section addresses the translation of RDF documents and covers each case from Section 3.3 in its own subsection.

4.1 General Mapping Format

The mapping is defined in XML with the root element `<mappings>`. Directly below this root element follows a variable number of `<namespace>` elements and at least one `<subject-group>` element. The `<namespace>` elements are used to define namespace prefixes as described further in Section 4.2, while the `<subject-group>` elements define groups of mappings. Within such a group every mapped property refers to the same subject. In other words, we create a subject group for every subject present in a source ontology and are consequently able to list properties and their mapping separately for each subject. There exist three different kinds of mappings. First, there is the simple mapping denoted through a `<simple-mapping>` element that contains only exactly one `<source>` and exactly one `<target>` element. The values of those elements have to be real properties from the source and target ontology respectively, the use of property functions is not allowed. However, it is possible to apply conversions with the help of the XML attributes explained in the Sections 4.3.2 and 4.3.6. The second type of mapping is called `<nested-mapping>` and is used if the properties we want to map are nested in the source or target ontology or both. The third form is the complex mapping represented as the XML element of the same name. It is used in connection with custom property functions that probably even need arguments to accomplish their task. These cases are different because they provide a great deal of flexibility with the downside that it is not possible to reverse these mappings in a generic way, which means if a reversion is possible and desired, it has to be specified explicitly. Besides the variable amount of these different mapping elements, a subject group must contain a `<source-type>` and a `<target-type>` element that declare the class type of the subject in the form of the *rdf:type* property. Both types are mandatory as they act as the main selection triple in

the generated SPARQL queries. A complete definition in XML Schema of this mapping language is given in Appendix B.1.

4.2 Namespaces

In RDF as well as in SPARQL queries we can use *XML Namespaces* to shorten the names of XML elements. Therefore, we define a prefix that can be used instead of the common part of the element URI. During processing of this data the prefix gets substituted back with the original URI part. In order to provide the same kind of abbreviation in our mapping language, we need the ability to define namespaces and their prefix as well. Listing 4.1 shows an example of how this mechanism is implemented in our mapping language. The syntax specifies for every namespace a `<namespace>` element with a `prefix` attribute. That attribute holds the name of the used prefix and the namespace URI to substitute is stored as the value of the element. Our example defines two namespaces with the respective prefixes *rdf* and *foaf*.

```
<?xml version="1.0"?>
<mappings>
  <namespace prefix="rdf">
    http://www.w3.org/1999/02/22-rdf-syntax-ns#
  </namespace>
  <namespace prefix="foaf">
    http://xmlns.com/foaf/0.1/
  </namespace>
</mappings>
```

Listing 4.1: Example mapping document with namespaces

The namespaces defined in this place are not only used to name the entities in the mapping file but also for naming purposes in the generated SPARQL query. Thereby every `<namespace>` element is translated into a PREFIX expression in SPARQL, whereas the name of the prefix is taken from the value of the `prefix` attribute in the mapping file and the namespace value from the value of the XML element. In our example document from Listing 4.1, this yields in the SPARQL fragment depicted in Listing 4.2.

```
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
```

Listing 4.2: Example of resulting SPARQL document with namespaces

The forward as well as the backward mapping use the same namespaces and prefixes, therefore this part is inserted the same way in both queries.

4.3 Translation of RDF Documents

This section addresses the cases defined in Section 3.3 and shows how they are represented and processed in our mapping. These examples demonstrate each case separately in a simple example. A real mapping with multiple cases appearing together would enclose each generated triple in an OPTIONAL pattern, so that an RDF resource can match even if it contains not all elements defined in the mapping. Likewise, if a mapping contains more than one subject group, one query

is generated for each of them. In order to focus on the essential aspects, we keep the translation simplified in this section, but we will present a complete and real example in Chapter 7.

4.3.1 Simple One to One Mapping

The simple one to one mapping is represented with a simple mapping as introduced in Section 4.1. It consists of exactly one `<source>` and exactly one `<target>` element and in this case without attributes. Listing 4.3 shows a fragment of a mapping document with such a mapping. In order to focus on the actual mapping, only the relevant part of the file is shown in the example. Namespace definitions and the embedding in a subject group are left out and would need to be added for a complete and valid mapping document.

```
...
<simple-mapping>
  <source>vCard:FN</source>
  <target>foaf:name</target>
</simple-mapping>
...
```

Listing 4.3: Example mapping file fragment with simple one to one mapping

```
# Namespace definitions omitted

CONSTRUCT {
  ?subject foaf:name ?name .
}
WHERE {
  ?subject vCard:FN ?name .
}
```

Listing 4.4: Resulting SPARQL fragment with simple one to one mapping

First, we focus on the generation of the forward mapping from the definition in Listing 4.3. The `<source>` element refers to the property from the source ontology we intend to map, therefore its value is placed in the WHERE clause of the SPARQL query generated for the mapping as shown in Listing 4.4. The `<target>` element on the other hand refers to the corresponding property from the target ontology, which means its value belongs in the CONSTRUCT clause of the query. In this simple case, we only want to replace the property, the subject and object values remain the same, which is why we bind those values to variables in the WHERE part and use them unaltered and in the same position in the CONSTRUCT part.

For the creation of the backward mapping query, all we need to do in this easy case is to swap the triples in the WHERE and CONSTRUCT clause. In a backward mapping, the value of the `<target>` element acts as the source and therefore is placed in the WHERE clause, likewise the `<source>` elements describes the new target, which means it belongs in the CONSTRUCT part of the query.

4.3.2 Untyped to Typed Mapping

In order to support the mapping from untyped to typed literals, we need to add a small extension to the simple mapping as used in the prior section. In this case, we also map one property onto another with the distinction that we add a datatype to the object as well. The consequences for our mapping language are small, we only need to declare what datatype the target has to use. We do this with an XML attribute named `datatype` belonging to the `<target>` element. As the backward mapping is generated from the same mapping definition, we can specify a *datatype* attribute in the `<source>` element too. This datatype is used if the object of the source property is also typed and this information needs to be preserved for the backward mapping. Therefore,

we are not only able to add datatypes to untyped literals but also to change already existing ones. The value of this attribute must be an URI string referencing the datatype to use. Listing 4.5 shows the significant fragment of an example mapping file. The source ontology in this example represents birthdays as untyped literals, whereas the target ontology uses the XML Schema type *dateTime* which is indicated in the `datatype` attribute. Additionally, let us assume the syntactical representation of birthdays do not match in the source and target ontology. This means, we have to convert them with the help of a converter class that is specified with its fully qualified name in the `datatypeConverter` attribute. This class provides a method to transform forwards and backwards between the different syntax forms and it must be on the Java classpath as it is dynamically loaded during execution.

```
...
<simple-mapping>
  <source>p:birthday</source>
  <target
    datatype="xsd:dateTime"
    datatypeConverter=
      "ch.uzh.ifi.bday">
    ex:bday
  </target>
</simple-mapping>
...
```

Listing 4.5: Example mapping file fragment with untyped to typed mapping

```
# Namespace definitions omitted

CONSTRUCT {
  ?subject ex:bday ?bday .
}
WHERE {
  ?subject p:birthday     ?b .
  ?b       fn:datatype
             xsd:dateTime ;
           fn:datatypeConverter
             "ch.uzh.ifi.bday" ;
           fn:addDatatype ?bday .
}
```

Listing 4.6: Resulting SPARQL fragment with untyped to typed mapping

The SPARQL query in Listing 4.6 represents the forward mapping and it is generated from the mapping definition in Listing 4.5. On first sight, it may look confusing that an extension to the `<target>` element has an impact on the WHERE clause and not on the CONSTRUCT. The reason for that lies in the SPARQL grammar which only allows variable bindings in the WHERE clause. In the CONSTRUCT part, we can only read the values of variables to build new RDF graphs. Therefore, we need to use property functions that create new bindings in the WHERE part. In our example, this means that the generated query first binds the object value of the *p:birthday* property to a temporary variable *?b*. This variable is then used as the subject for our property functions described in Section 3.3.2 that add a datatype to a literal. As already explained, this happens in three steps. First, the datatype is set with the *datatype* property function and the value from the `datatype` attribute as object. Then, the converter class is specified with the *datatypeConverter* property function and the value from the `datatypeConverter` attribute. At last, the *addDatatype* function is called and binds the new typed and converted value to the variable passed as object. After that, this variable (*?bday*) can be used in the CONSTRUCT clause to create the new document expressed in the target vocabulary. The usage of these property functions happens transparently for the user, so that the mapping file and the resulting RDF document are free from property functions.

The generation of the backward mapping is not as simple as in the prior case. It depends on the `<source>` element and if it has a `datatype` attribute as well. If so, the generated query is like the forward query with the predicates of the main triples exchanged and the object of the

fn:dataype function is replaced with the corresponding value from the `datatype` attribute of the `<source>` element. However, if the `<source>` element is untyped and therefore does not feature a `datatype` attribute, we need to remove the existing datatype in a backward mapping. For this, we introduce the property function *fn:removeDatatype* that acts as a counterpart to the *fn:addDatatype* function in the way that it takes the input subject, strips its datatype, and then assigns it to the object variable. The source code of this property function is shown in Appendix A.2. Listing 4.7 shows the backward query for the mapping from Listing 4.5. The predicates of the main triples are swapped and the datatype converter triple stays the same as each such class must work for both ways of conversion. As the most important difference is to mention that *fn:datatype* is not used and in the last triple of the WHERE clause we use the *fn:removeDatatype* property function instead.

```
# Namespace definitions omitted

CONSTRUCT {
  ?subject p:birthday ?bday .
}
WHERE {
  ?subject ex:bday              ?b .
  ?b       fn:datatypeConverter "ch.uzh.ifi.bday" ;
           fn:removeDatatype    ?bday .
}
```

Listing 4.7: Resulting SPARQL fragment with untyped to typed backward mapping

4.3.3 Extracting Nested Data

The simple mapping is not adequate to support nested data, because we need to describe properties that consist of other properties instead of value objects that can be copied directly. Therefore, we introduce nested mappings and the concept of containers in our mapping language. There exist two kinds of containers, one for structured elements in the source and one for the same in the target ontology. Containers feature a name and an optional type that are represented as the attributes `name` and `type` in the mapping. Every container consists of a variable number of subelements that are either `<source>` or `<target>` elements depending on the kind of container. In this example (Listing 4.8), we have nested data in the source ontology, which means we need a source container that is the container named *vCard:N*. Our target ontology is flat and therefore the `<target>` elements are listed outside any container before the `<source-container>` element. The number of `<source>` elements must be equal to the number of `<target>` elements and they are matched according to the order in that they are listed. In our example, this means the first `<source>` element (*vCard:Family*) is mapped to the first `<target>` element (*foaf:family_name*) and the second source (*vCard:Given*) to the second target (*foaf:firstName*).

Listing 4.9 shows the generated SPARQL query used for forward translations for this example mapping. The two `<target>` elements map to two elementary triples in the CONSTRUCT clause. The WHERE part of the query is more complex due to the presence of a source container in the mapping. Such a container maps onto a main triple with the current subject, the predicate named by the `name` attribute of the container, and a variable as object. This variable is bound to a blank node that connects the property with additional secondary properties. These are the `<source>` elements contained in the container and they are each mapped to a triple with the object from the main triple as subject, the value of the respective `<source>` element as predi-

```
...
<nested-mapping>
  <target>foaf:family_name</target>
  <target>foaf:firstName</target>
  <source-container name="vCard:N">
    <source>vCard:Family</source>
    <source>vCard:Given</source>
  </source-container>
</nested-mapping>
...
```

Listing 4.8: Example mapping file fragment for extracting nested data

```
# Namespace definitions omitted

CONSTRUCT {
  ?subject foaf:family_name ?fam ;
           foaf:firstName ?first .
}
WHERE {
  ?subject vCard:N      ?n .
  ?n       vCard:Family ?fam ;
           vCard:Given  ?first .
}
```

Listing 4.9: Resulting SPARQL fragment for extracting nested data

cate, and a variable as object. Only these object variables are finally used to build the translated document, the other ones are solely used to match the structure in the source document.

Although not used in this example, the `<source-container>` element could have an optional `type` attribute. This would act as the class type of the connection object and would add another triple with the connection object as subject, *rdf:type* as predicate, and the value stated in the `type` attribute as the object.

In the generation of the backward query, the roles of the `<source>` and `<target>` elements are reversed. If we closely analyze this situation, we come to the conclusion, that we now have unnested sources (the original `<target>` elements) and targets nested in a container (the original `<source>` elements). This scenario fits exactly the requirements for the case 'create substructures' described in the next section. Hence, the backward mapping in this case corresponds to the forward mapping of the next case with inverted sources and targets. We therefore refer the reader to the next section to get the full details of that query generation.

4.3.4 Create Substructures

In the case of a flat source ontology and a nested target structure, we need the other kind of container called target container. Its usage is the same as with the source container with the obvious difference that the subelements are `<target>` instead of `<source>` elements. As we see in Listing 4.10, the unnested `<source>` elements are listed before the `<target-container>` element and they match the enclosed `<target>` elements in number and position. Of course, this means that we use a nested mapping again.

This time, the `<source>` elements are flat and therefore are mapped onto two plain triples in the WHERE clause for the forward mapping. Both triples consist of the current subject, the respective predicate, and for each a variable as object that will be used in the CONSTRUCT part. The target container is mapped similar to the source container in the prior example with the generated triples appearing now in the CONSTRUCT clause. The container itself is translated into a triple with the current subject, the name of the container taken from the `name` attribute, and a blank node identifier. Although this blank node identifier is always the same in the query, SPARQL generates a new one for every distinct subject. From this point on, the rest of the generated triples for this mapping have the blank node as subject. In our example, the container has a `type` attribute set that translates into a triple with an *rdf:type* predicate and the value from the

```
...
<nested-mapping>
  <source>
    foaf:family_name
  </source>
  <source>
    foaf:firstName
  </source>
  <target-container name="vc:n"
      type="vc:Name">
    <target>
      vc:family-name
    </target>
    <target>
      vc:given-name
    </target>
  </target-container>
</nested-mapping>
...
```

Listing 4.10: Example mapping file fragment for creating substructures

```
# Namespace definitions omitted

CONSTRUCT {
  ?subject vc:n      _:n .
  _:n rdf:type       vc:Name ;
      vc:family-name ?fam ;
      vc:given-name  ?first .
}
WHERE {
  ?subject foaf:family_name ?fam ;
           foaf:firstName   ?first .
}
```

Listing 4.11: Resulting SPARQL fragment for creating substructures

`type` attribute as object. After that, each subelement is mapped to a triple with the same variables as objects that were bound in the WHERE part. The resulting SPARQL query fragment is depicted in Listing 4.11.

Inversely to the case described in Section 4.3.3, the generation of the backward query of this mapping corresponds to the creation of the forward query of the prior case 'extracting nested data', again with reversed roles for sources and targets. Therefore, the reader is referred to the previous section for the details.

4.3.5 Converting Structures

This example, as shown in Listing 4.12, deals with nested structures in both the source and the target ontology. This translates to a nested mapping with exactly one `<source-container>` and one `<target-container>` element in our mapping. The containers are the same as in the last two examples, each containing the same number of subelements that get matched in the order of their appearance. Both containers need one `name` attribute and can optionally have one `type` attribute.

The forward mapping to the query shown in Listing 4.13 is generated the same way like the respective parts in the previous two examples. The source container is translated into a triple structure in the WHERE clause as explained in Section 4.3.3 and the target container is transformed into triples of the CONSTRUCT part as described in Section 4.3.4.

The backward mapping query is again generated with reversed roles for the containers as well as the `<source>` and `<target>` elements. This means, the source container acts as the target container, the sources as targets, and vice versa. Under those premises, the backward query is

```
...
<nested-mapping>
  <source-container name="vc01:N">
    <source>vc01:Family</source>
    <source>vc01:Given</source>
  </source-container>
  <target-container name="vc06:n"
      type="vc06:Name">
    <target>vc06:family-name</target>
    <target>vc06:given-name</target>
  </target-container>
</nested-mapping>
...
```

Listing 4.12: Example mapping file fragment for converting structures

```
# Namespace definitions omitted

CONSTRUCT {
  ?subject vc06:n _:n .
  _:n rdf:type    vc06:Name ;
      vc06:family-name ?fam ;
      vc06:given-name  ?first .
}
WHERE {
  ?subject vc01:N      ?n .
  ?n       vc01:Family ?fam ;
           vc01:Given  ?first .
}
```

Listing 4.13: Resulting SPARQL fragment for converting structures

generated like its forward counterpart.

4.3.6 Literals to URIs

Mappings belonging to this category are simple mappings with the extension that we need to convert the object from a literal value into an URI. This means, in addition to the source and target property, we need to name the Java class responsible for the conversion, so that we can use the property function defined in Section 3.3.6 to perform the transformation. This information is stored in the `uriConverter` attribute of the `<target>` element. Its value must be the fully qualified name of a Java class that satisfies all requirements defined in Section 3.3.6. In our example in Listing 4.14, this converter class is *ch.uzh.mailto* and it must be present on the Java classpath.

```
...
<simple-mapping>
  <source>swrc:email</source>
  <target
      uriConverter="ch.uzh.mailto">
    foaf:mbox
  </target>
</simple-mapping>
...
```

Listing 4.14: Example mapping file fragment for literal to URI mapping

```
# Namespace definitions omitted

CONSTRUCT {
  ?subject foaf:mbox ?email .
}
WHERE {
  ?subject swrc:email    ?m .
  ?m       fn:uriConverter
             "ch.uzh.mailto" ;
           fn:convertURI ?email .
}
```

Listing 4.15: Resulting SPARQL fragment for literal to URI mapping

Listing 4.15 depicts the generated SPARQL query for the forward part of this mapping. We

notice that the whole transformation of the object happens in the WHERE clause, as this is the only valid place where we can bind variables to new values. The `<source>` element is mapped to a triple in the WHERE part that retrieves the original literal and stores it in a temporary variable (?m) which acts then as the source for our self-defined property functions. First, a triple is generated that makes a call to the *uriConverter* function passing the value from the `uriConverter` attribute as object. The subject in this triple is the variable with the literal we want to convert. This is not absolutely necessary, because the property function only uses the object of the triple, but it makes the mapping clearer and more readable. In a second step, another triple is generated that invokes the *convertURI* property function. This time, the correct subject is needed because the function uses its value for the conversion and thereafter binds the outcome to the variable passed as object. The same variable is also used as the object in the only triple of the CONSTRUCT clause to build the target document.

The property function *uriConverter* contains methods for both the forward and the backward conversion of the input, which means this triple is also present in the backward query. Likewise, the *convertURI* function is used the same way in both queries as it is build to convert between literals and URIs depending on the kind of the input as described in more detail in Section 3.3.6. The only change for the backward query is once again the exchange of the predicates in the main triples.

4.3.7 Restoring Implicit Knowledge

In this case, we introduce the third kind of mapping: the complex mapping. The requirements of this case exceed the capabilities of the simple mappings even with the already introduced extensions. Some kind of nesting is also not given, consequently we cannot use a nested mapping and thus need the complex mapping. In such a mapping, we can use self-defined property functions with the option of also passing arguments to them. This flexibility comes at the price that it is no longer possible to reverse the mapping in a generic way. This means, we need to define the mappings in both directions explicitly. In our mapping language, this is done with the forward mapping for RDF documents enclosed in exactly one `<forward-mapping>` element and the backward mapping in at most one `<backward-mapping>` element, if a backward mapping is possible and desired.

Our example in Listing 4.16 has a backward mapping and therefore its `<complex-mapping>` element consists of one `<forward-mapping>` element and one `<backward-mapping>` element. Each of them contains two `<arg>` elements used to pass arguments to the property function invoked later as well as one `<source>` and one `<target>` elements. In both mappings, the `<target>` element names the property whose value we want to compute and the `<source>` element states the property function to use for that purpose.

Albeit the more complex mapping definition, the generated SPARQL query (Listing 4.17) for the forward mapping is not that complicated. In the WHERE clause, we first need to get the objects of all arguments we want to pass, which results in triples with the current subject, the properties from the `<arg>` elements, and variables as objects. After all arguments are bound to variables in this way, we invoke the *args* property function with a list of the just bound variables. The last step in the WHERE part is to call the *toDuration* property function that uses the passed arguments to compute the outcome and binds it to the variable placed as object. The CONSTRUCT clause is straightforward with only one triple that uses the common subject, the property from the `<target>` element in the mapping, and as object the variable that was bound by the *toDuration* function.

The backward part of a complex mapping is generated based on the `<backward-mapping>` in the same way as the forward mapping. The CONSTRUCT clause contains only one triple with the predicate listed as the target in the mapping. In the WHERE part, we first bind the

```
...
<complex-mapping>
  <forward-mapping>
    <arg>prop:Start_date</arg>
    <arg>prop:End_date</arg>
    <source>f:toDuration</source>
    <target>ex:duration</target>
  </forward-mapping>
  <backward-mapping>
    <arg>ex:start</arg>
    <arg>ex:duration</arg>
    <source>f:toEndDate</source>
    <target>prop:End_date</target>
  </backward-mapping>
</complex-mapping>
...
```

Listing 4.16: Example mapping file fragment for restoring implicit duration knowledge

```
# Namespace definitions omitted

CONSTRUCT {
  ?subject ex:duration ?duration .
}
WHERE {
  ?subject prop:Start_date ?start;
       prop:End_date   ?end ;
       fn:args (?start ?end) ;
       f:toDuration    ?duration .
}
```

Listing 4.17: Resulting SPARQL fragment for restoring implicit duration knowledge (forward)

objects matching the predicates from the `<args>` elements to variables that are then passed as arguments to the *args* property function. At last, the function named by the `<source>` element is called which computes the result and assigns it to the object variable. Listing 4.18 shows the resulting query for the backward mapping from Listing 4.16.

```
# Namespace definitions omitted

CONSTRUCT {
  ?subject prop:End_date ?end .
}
WHERE {
  ?subject ex:start    ?start ;
           ex:duration ?duration ;
           fn:args     (?start ?duration) ;
           f:toEnd     ?end .
}
```

Listing 4.18: Resulting SPARQL fragment for restoring implicit duration knowledge (backward)

Needless to say that a backward query can and is only generated if a complex mapping contains a `<backward-mapping>` element in its definition.

4.3.8 Substitution of Class Types

As identified in Section 3.3.8, this case cannot be handled with one of the other mappings, because we replace the object and not the predicate of a triple. Additionally, this case is only applied to

```
...
<subject-group>
  <source-type>
    vCard:VCard
  </source-type>
  <target-type>
    foaf:Person
  </target-type>
  ...
</subject-group>
...
```

Listing 4.19: Example mapping file fragment for type substitution

```
# Namespace definitions omitted

CONSTRUCT {
  ?subject rdf:type foaf:Person .
}
WHERE {
  ?subject rdf:type vCard:VCard .
}
```

Listing 4.20: Resulting SPARQL fragment for type substitution

a special kind of triples where the predicate is always known in advance and the same, namely *rdf:type*.

Listing 4.19 shows a fragment of a mapping file with the definitions for the source type and the target type. The types are defined once for every subject group and their definitions are placed right after the opening tag of a new subject group before any other mappings are defined.

Those type definitions are translated into a SPARQL query for the forward query as depicted in Listing 4.20. The value of the `<source-type>` element is placed as the object in a triple in the WHERE clause of the query. The triple consists of the subject given from the current subject group and the predicate *rdf:type*. The value of the `<target-type>` on the other hand serves as the object for an identical triple in the CONSTRUCT clause.

In the generation of the backward query, the source and target types reverse their roles. This leads to a query like in the forward case with the difference that the objects of the *rdf:type* triples from the CONSTRUCT and WHERE clauses are exchanged.

Chapter 5

Architecture of the RDF Transformer

In this chapter, we present the architecture of the RDF Transformer and give an overview of its usage. We describe the individual components and how they interact with each other to perform the functionality of transforming RDF data from one ontology into another. Please note that this chapter covers only a bird's eye view on the individual components, their detailed structure and implementation is described in Chapter 6. Further, we highlight the programming interface for the user with all the possibilities to interact with the RDF Transformer. Figure 5.1 shows this architecture and overview in a diagram.

The component called *RDF Transformer* depicted in the center of the figure is the main component of the application. It is responsible for controlling and calling the other components during the process of a transformation. For this purpose, it offers a programming interface to the user, but in order to fulfill its duty it needs mappings which must be defined and introduced in advance. The management of those mappings is the task of the *Mapping Storage* that we inspect first.

The *Mapping Storage* component is responsible for all aspects concerning mappings. In a first step, this includes offering a programming interface for users that enables them to register new mappings. This operation is labeled as step one 'registerMapping' in Figure 5.1. For the registration of a new mapping, a user must submit a mapping definition file as defined in Chapter 4 and Appendix B.1, the namespace URIs of the source and target ontologies, and optionally a JAR file containing custom property functions, if needed by the mapping. As this JAR file contains executable code from a possible untrusted third party, it would not be safe to just run the classes in it. Therefore, we execute such code in a secured environment given by the Java security model as explained more precisely in Section 6.7. After the mapping storage receives these inputs, it generates the SPARQL queries that are used for the actual transformation of RDF data. Normally, this results in a set of queries for both the forward and backward translation. As the generation of these queries is a time consuming process and needs not to be repeated on every run of the application, the mapping storage saves these mappings to disk. This also allows the user to first preload the mapping storage with a series of mappings and then just use the transformer to translate multiple documents. After this digression to the mapping storage, we can now go back and take a closer look at the *RDF Transformer* component.

As already stated, the *RDF Transformer* component is the controller of the whole application. Its programming interface follows loosely the one of the *Model* class in the Jena Semantic Web Framework. This means, it consists of three methods (with some overloaded variants) that determine the course of the transformation process. These methods are *read*, *transform*, and *write*. Their roles are explained in the following paragraphs. To start a transformation process the transformer

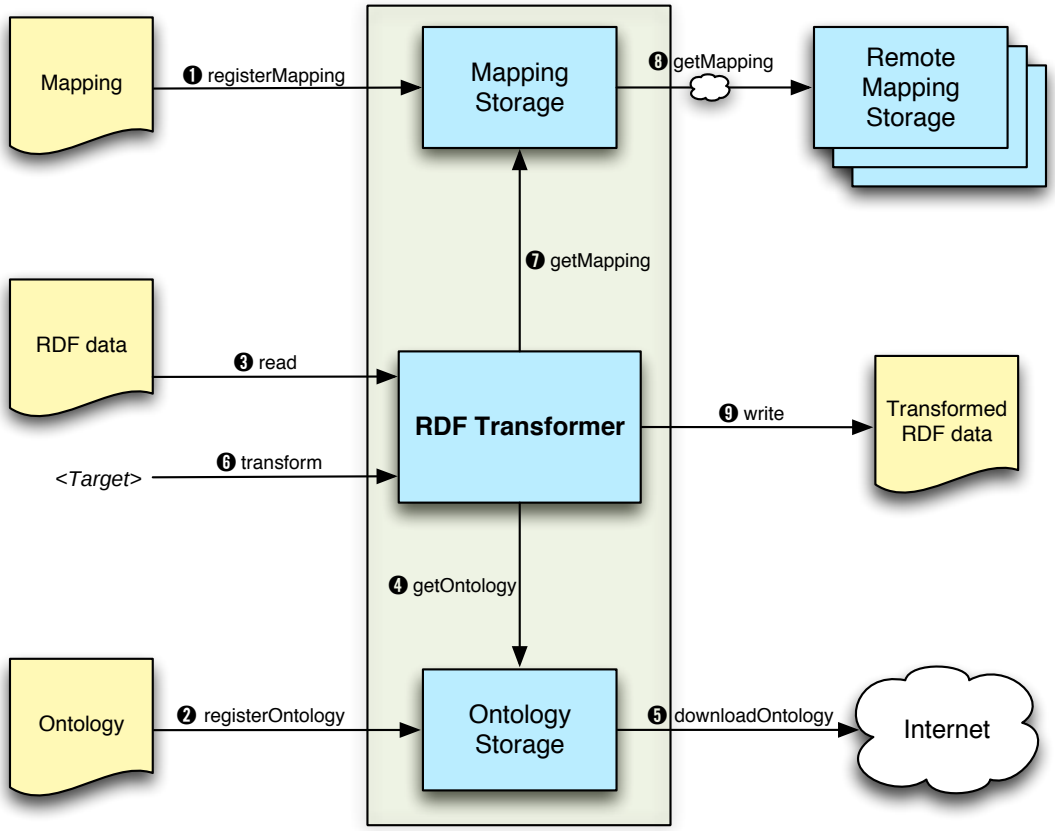

Figure 5.1: Overview and architecture of the RDF Transformer

first needs the RDF source data as input. That is obtained with the first of the three methods, the *read* function labeled as step three in Figure 5.1. Its responsibility is to read the data from the given input stream and construct a Jena Model from it. We use a generic input stream as argument to increase the flexibility of the application as we can apply the same method regardless of the origin of the input being a file, the memory, or the network. For the syntax, users can use the same forms supported by the Jena Semantic Web Framework as we use Jena to create the model. In a next step, the transformer tries to infer more information from the read data with the help of a reasoner. In order to accomplish this, the reasoner needs the ontology definitions of the involved vocabularies. The acquisition and management of those is the task of the *Ontology Storage* component. This component is similar to the *Mapping Storage* as it handles every aspect that has to do with ontologies. The transformer requests the relevant ontology definitions from the ontology storage (shown in Figure 5.1 as step four 'getOntology') and if the ontology could be received, it uses its definition to apply the reasoner to the source data. Typically, the result of this operation contains more data, whereof the type information is at most importance to us, because it is later used as the main selection criterion in the transformation. Before we continue with the description of the *RDF Transformer* component, let us examine the *Ontology Storage* component a little bit closer.

The main function of the *Ontology Storage* component is to provide and store ontology definitions. It can obtain those on two different ways. The first and mostly preferred way is to let the

ontology storage download the ontology file automatically from its namespace URI (see step five in Figure 5.1). It belongs to the best practices for publishing RDF vocabularies that its ontology definition can be obtained this way [MBS08] and most ontology designers adhere to this. If the automatic download for an ontology definition is not possible or not desired, there is a second way of ontology registration. Users can manually register ontologies prior to the reading of the input data via the 'registerOntology' operation labeled as step two in Figure 5.1. In this case, the user must submit an input stream to the ontology definition that gets read into the application. After an ontology definition is downloaded from the Internet or read from an input stream, it is stored on disk so that it can be used in further runs of the application, without the need for registering and obtaining it again.

After the RDF source data is loaded into the transformer and the reasoning is completed, the application is ready to transform this data into any target ontology, if a suitable mapping is available. A transformation is triggered with a call by the user to the *transform* operation labeled as step six in Figure 5.1, which takes the desired target ontology as argument. Before it can request mappings from the mapping storage, it needs to determine which ontologies are actually used in the source data. To do this, it analyzes the source data and extracts every namespace used which usually leads to a whole list of source ontologies. For each namespace URI from this list and the user submitted target ontology, the transformer queries the mapping storage for an applicable mapping (step seven in Figure 5.1). If no match is made on the local mapping storage, the *Mapping Storage* has the possibility to query remote mapping storages, which is depicted as step eight in Figure 5.1. There can be an arbitrary number of such remote storages and they are queried one after another until an appropriate mapping is found or there are no remote mapping storages left. In that case, a transformation of this source ontology is not possible with the current mappings available and it is continued with the next source-target pair. The addresses of these remote mapping storages are defined in the configuration file (described in Section 6.7) of the RDF Transformer. It can contain no entries at all, in which case the querying of remote mapping storages is disabled or if it contains entries, they are called in the order they appear in the file. If a matching mapping is retrieved from a remote mapping storage, it is cached locally to serve further request of the same kind.

The *Remote Mapping Storages* are self-contained servers on the Web that hold mappings for RDF Transformers. They cannot create the mapping queries themselves, but they offer mappings generated in advance by one RDF Transformer to others. It is also possible to add new (previously generated) mappings to the remote mapping storages at runtime. These servers are queried over HTTP and return the mapping as a ZIP file containing queries and optional property function code in a HTTP response to the caller.

Every matching mapping that is found, locally or remote, is applied on the source data and the results are continuously merged into one combined model that in the end constitutes as the final result. This result is deposited in the RDF Transformer until the user makes a call to the *write* operation that enables him to retrieve the result model (step nine in Figure 5.1). This write method expects two arguments: an output stream to write to and a syntax form for the output. The use of a generic output stream permits like in the read operation a great flexibility as one single method can be used to write to files, memory, or the network. In the same way, every syntax form supported by the Jena Semantic Web Framework can be used in the write function as the actual writing is again delegated to Jena.

At this time of execution, the original source data and the transformed target data are still stored in the RDF Transformer. This implies that we can retrieve the target data again without the need to transform the source data anew. We can simply call the write method again, even with a different value for the syntax form argument. Likewise, we can transform the source data into other target data without the need to repeat the reading and reasoning of the input, but note that in this case the previously transformed data is overwritten. Of course, it is also possible to read

new source data and start over with a new transformation.

Chapter 6

Implementation of the RDF Transformer

In the last chapter, we presented an overview of the RDF Transformer, its components, and how they work together. In this chapter, we focus on the individual components and their implementation. For that purpose, we use the same segmentation of components as already identified in Chapter 5, namely *RDF Transformer*, *Mapping Storage*, *Remote Mapping Storage*, *Ontology Storage*, and in addition the *SPARQL Extensions* that were already shown in parts in Section 3.3, as well as the configuration file and security policy.

6.1 Package Overview

Figure 6.1 shows an overview of all packages composing the RDF Transformer. Each package is addressed in its own section in the remainder of this chapter.

6.2 RDF Transformer

The central class of the RDF Transformer component is the class of the same name. It provides the programming interface for this module to the user and controls the action of the other classes. We will first introduce these other classes and explain afterwards how the *RDFTransformer* class makes use of them. All classes belonging to this component are depicted in the class diagram in Figure 6.2.

TestConsole

The RDF Transformer is not intended to be used as a standalone application but as a library linked into other programs, therefore it would normally not provide a user interface for direct interaction. For testing purposes, we created a class called *TestConsole* that provides a simple command line interface for the RDF Transformer that enables us to test every aspect of the transformer without the need of embedding it into a full program. Therefore, this is an executable class that reads a set of arguments from the command line, analyzes them, and finally calls the appropriate methods of the RDF Transformer. The syntax for these command line arguments is described in Appendix C.1.

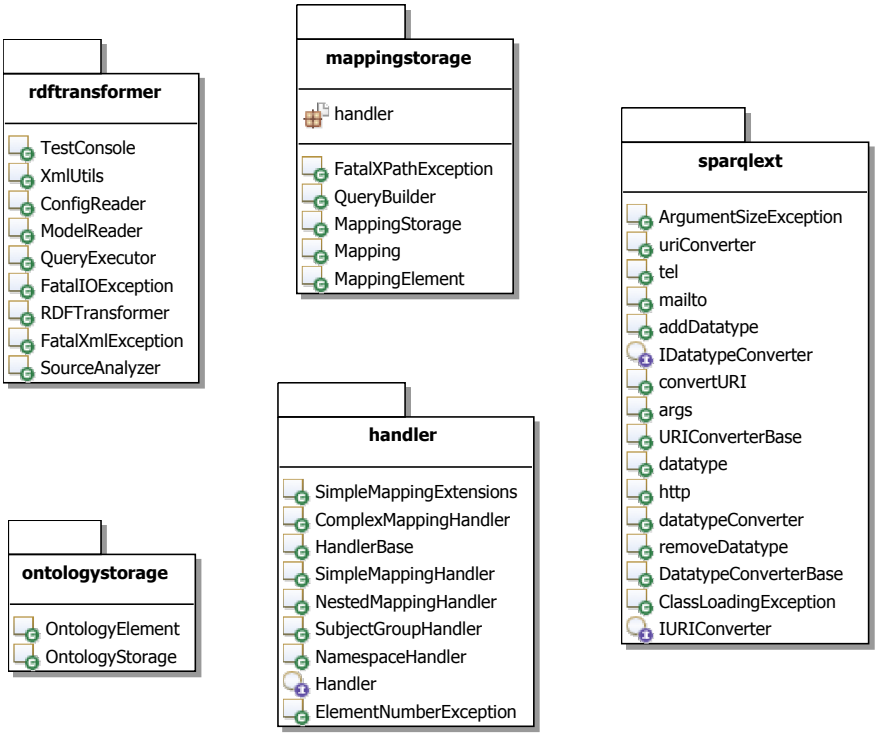

Figure 6.1: Package overview of the RDF Transformer

ConfigReader

The *ConfigReader* class is used as the central access point to the configuration of the whole application. It is responsible for reading the configuration file described in Section 6.7, storing the settings internally, and providing accessor methods to retrieve them. Every other part of the transformer uses this class if it needs the value of a configuration option. This decreases the access time as not every module needs to read the configuration file from disk itself and it enables us to control all application settings in one location. Hence, this class is implemented as a Singleton to ensure that only exactly one instance exists at all time.

ModelReader

Before we can start a transformation, the RDF Transformer needs input data. The reading of this data falls under the responsibility of the *ModelReader* class and more precisely its public, static *read* method. It can read data from a variety of sources as it takes a generic input stream as argument for this purpose. Likewise, it supports for this input data every syntax known to the Jena Semantic Web Framework trough its third argument. With the second argument, it is possible to specify a base URI used for converting relative URIs to absolute ones in the input data. The *ModelReader* class uses these arguments in conjunction with Jena to read the RDF data in the given syntax from the input stream and to construct a Jena model containing the source data. In a next step, it tries to enlarge this data with inferred information generated with the Pellet[1] reasoner. For

[1] http://pellet.owldl.com

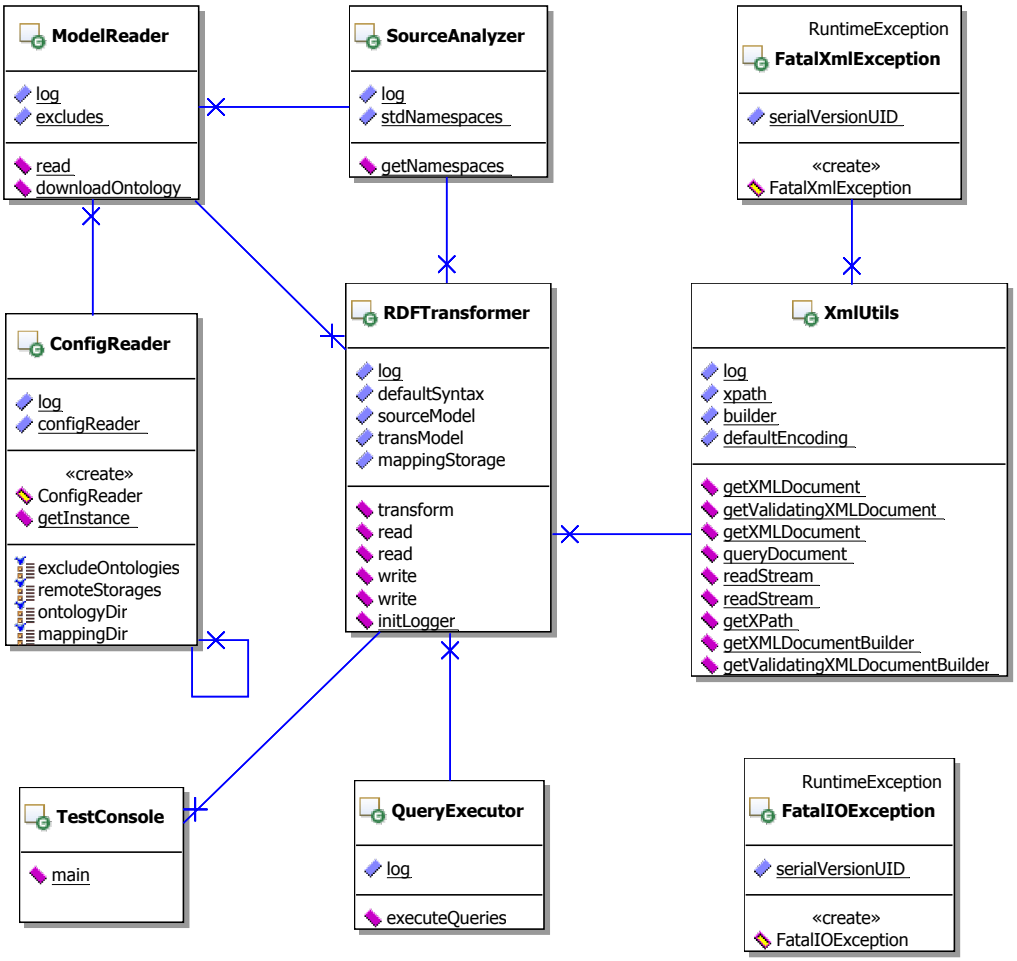

Figure 6.2: Class diagram of the RDF Transformer component

this, we first need to know what ontologies are used in the source data in order to obtain the right ontology definitions. Therefore, the source data is examined by the *SourceAnalyzer* class that determines the ontologies used. That class is described in the following section. After the *ModelReader* has a list of used ontologies, it queries the ontology storage for each of its definitions. If a definition is found that way, it is passed together with the source data to the reasoner. If no definition is available locally the *ModelReader* tries to download the ontology definition from the Web with the namespace URI as location address, if it is not on the list of ontologies to exclude from downloading. This list is defined in the configuration file and its purpose is explained in Section 6.7. After a successful download, the definition is added to the local ontology storage and used with Pellet or else no reasoning is done for this source ontology. These steps are repeated for every entry in the list of ontologies and the inferred models are all merged. Finally, this enriched model is returned to the caller and used as the source data.

SourceAnalyzer

The job of the *SourceAnalyzer* class is simple, therefore it is a rather short class with only one static method: *getNamespaces*. It takes a Jena model as argument and uses methods from the Jena framework to determine the namespace URIs of all ontologies used in that source model. From the resulting list it removes the standard namespaces found in every or most RDF resources that need no transformation. Currently, those are the namespaces of OWL, RDF, RDF Schema (RDFS), and XML Schema Datatypes (XSD). The remaining ontology URIs represent candidates for a mapping and are thus returned as a list.

QueryExecutor

The *QueryExecutor* class is applied much later than the classes described previously. After the source data is read and a matching mapping is found, the data (in the form of a Jena model) and the transformation queries from the mapping are passed to the *executeQueries* method of this class. First of all, the Jena model is converted into an ARQ DataSet object as the query execution classes operate with such objects and not with models. Thereafter, each of the queries is executed separately on the dataset through the ARQ SPARQL engine. The results are continuously merged and after the last query is processed returned to the caller.

XmlUtils

As its name already suggests, the *XmlUtils* class consists of a series of static helper methods for XML processing. Our mapping definition files are expressed in XML which consequently creates the need of such processing methods. There are three kinds of methods publicly available in this class with the first being two *readStream* methods that can be used for reading XML data from an input stream into a string. One takes an input stream and a string indicating the encoding of the stream as arguments, whereas the other omits the encoding parameter and uses the default encoding (UTF-8) instead. The second kind of methods is used for constructing DOM Document objects either from XML strings or from single DOM nodes. These are needed because most methods handling XML data expect this data as such an object, but we read and process them as strings. The method *getXMLDocument* converts a string to a DOM Document objects without validating the XML input. The *getValidatingXMLDocument* method does the same with validation against an XML Schema whose filename must also be submitted to the method. The schema file must be present in the '*schemas*' subdirectory of the application directory. There exists a second *getXMLDocument* method that expects a DOM Node object as argument and converts it, including all child nodes, into a full DOM Document. The third kind of methods is the *queryDocument* method that allows the caller to evaluate an XPath expression on a given DOM Document. The purpose of this is explained in more detail in Section 6.3, but let us just say here that XPath is used to process the mapping definition files.

FatalXmlException & FatalIOException

These two exceptions represent severe error conditions in our application. The *FatalXmlException* indicates either that the XML processing subsystem could not be loaded or that the parsing of an XML resource failed, which should not happen during normal operation and therefore marks a fatal exception. The *FatalIOException* is thrown if one of the required files on disk is not found or cannot be read, which prevents the application from successfully performing.

RDFTransformer

The main function of the *RDFTransformer* class is to provide a programming interface to the user and to control the course of the application. As already described in Chapter 5, the programming interface consists of the three methods *read*, *transform*, and *write*.

The *read* method takes an input stream, the base URI, and a string with the name of the syntax as arguments. There is also an overloaded version to support calls that do not specify a syntax, in which case the default syntax RDF/XML is expected. The methods delegate the read request to the *ModelReader* class and store the result in an internal variable.

The *transform* method expects a namespace URI of the desired target ontology as its argument. It first uses the *SourceAnalyzer* class to get the ontologies used in the source data and then queries the mapping storage for every combination of source and target ontology. If a mapping is found, it is applied by an instance of the *QueryExecutor* class or else that source is ignored. Each application of a mapping results in a transformed model that is merged successively into one final model. This model is also stored internally in the *RDFTransformer* class for further use.

The *write* method is the counterpart of *read* as it writes the result in the target ontology to an output stream with a syntax requested by the used or the default RDF/XML in the case of the overloaded version of this method. It uses functionality of the Jena Framework for the actual serialization of the model and thereby supports again the same syntaxes as Jena.

6.3 Mapping Storage

The *Mapping Storage* component consists of the *MappingStorage* class itself that acts as the programming interface of this component and its supporting classes. Those are the handler classes that process a mapping definition file and the *QueryBuilder* class that assembles the final transformation queries from individual parts generated by the handlers. As these handler classes are numerous and form their own package, we present them in their own section right after this one, but first we present the supporting classes and the main class *MappingStorage* as depicted in the class diagram in Figure 6.3.

QueryBuilder & Mapping

The task of the *QueryBuilder* class is to generate the transformation queries from a given mapping definition. It employs the handler classes described in Section 6.3.1 to generate the query fragments and assembles them to full queries. First, it uses the *XmlUtils* class to convert the string representation of the XML mapping file into a DOM Document object as described in Section 6.2. This Document object is then passed to an instance of the *NamespaceHandler* class that extracts its namespace definitions. Thereafter, the subject groups are separated and each of them is handled individually by a *SubjectGroupHandler* that performs the further processing. All handlers generate query fragments that are collected in two instances of the *Mapping* class (one for the forward and one for the backward mapping) that the *QueryBuilder* holds. After the handlers fully processed the mapping definition, the *assembleQuery* method builds one final transformation query for every subject group and mapping direction. Those are then stored in the class as well and can be retrieved through the respective accessor methods.

FatalXPathException

The *FatalXPathException* is thrown if one of the predefined XPath expressions fails to evaluate. Normally, this would only happen with malformed XPath code and since our expressions are

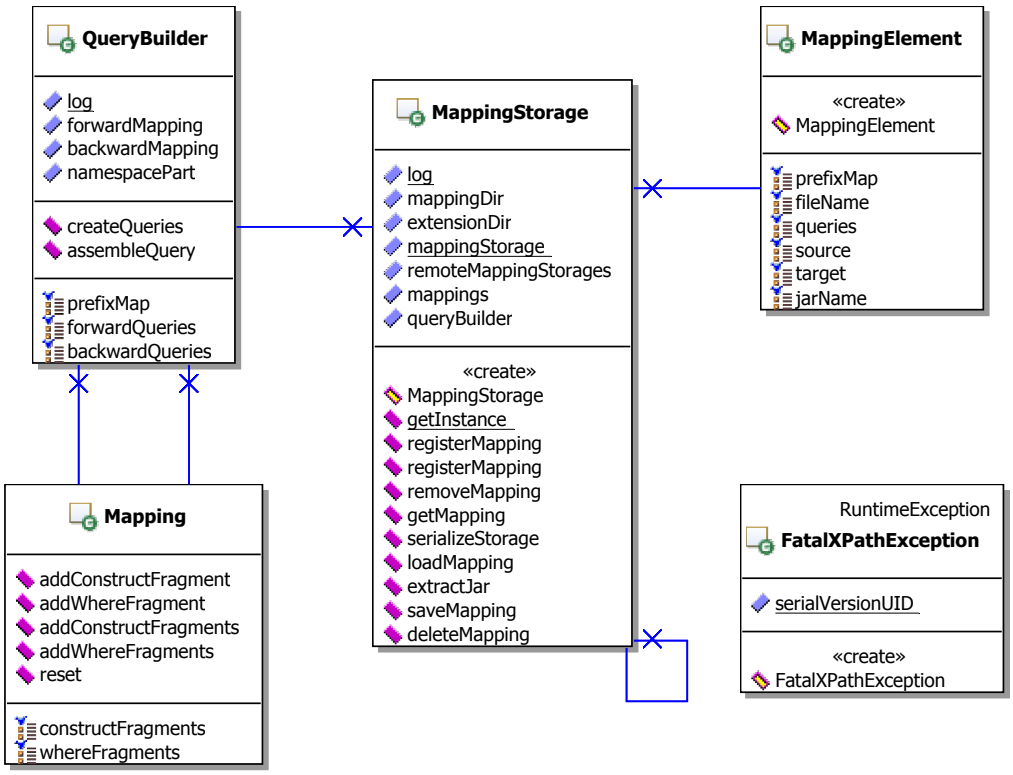

Figure 6.3: Class diagram of the Mapping Storage component

fixed this should not happen and would be a severe error situation indicated by this fatal exception.

MappingStorage & MappingElement

The mapping storage needs exclusive access to the mappings and therefore only one instance may exists at a time. To ensure this, the *MappingStorage* class is implemented according to the Singleton design pattern.

The responsibilities of the mapping storage are to administer mappings and to create new ones. When the *MappingStorage* instance is created, it loads a directory file into memory that contains the information on which mappings are already generated and available from the local file system. With this knowledge, it fills a hash map so that the actual mappings can be retrieved from disk on demand. The hash map uses the namespace URIs of the source and target ontolgies from the respective mapping as key and an instance of the class *MappingElement* as element. That class serves as an encapsulation with accessor methods for all the parts of a mapping. Those are a map with the prefixes and namespaces used in the mapping, the name of the file on disk the mapping is stored, the transformation queries, the namespaces URIs of the source and target ontology, and optionally the name of a JAR file that contains custom ARQ property functions used for this mapping.

The *MappingStorage* class offers the method *getMapping* to retrieve a previously registered mapping. For that, it needs the namespace URIs of the source and target ontology to identify the mapping. It then checks if such a mapping is available locally and returns the matching

MappingElement object on success. It is possible that the mapping must first be loaded from disk which is done by the *loadMapping* method. If no mapping is found and there are remote mapping storage servers defined in the configuration file, it continues its search on them, one after the other. For that, it establishes a connection to each server and requests the mapping until it is found or the last server returns a negative response. If a mapping is found remotely, it is added to the local storage, so that the next request for this mapping can be satisfied locally, and finally a *MappingElement* object from this mapping is returned to the caller. Adding a remote mapping to the local storage happens in three steps. Firstly, the mapping must be written to a file in the mapping directory on disk. This is done with the *saveMapping* method that saves the contents of the *MappingElement* object into a ZIP file. Secondly, the mapping directory file on disk must be updated which is done by the *serializeStorage* method that serializes the hash map to an XML file. Thirdly, if the mapping contains a JAR file with custom property functions, those must be extracted to the extension functions directory. The *extractJar* method is used for this and ensures thereby that those methods are on the classpath and can be found during query execution. Else, if no mapping could be found, the *getMapping* method reports this to the caller with a return value of 'null'.

Furthermore, there are two methods called *registerMapping* to register new mappings based on a mapping definition file. The only difference between the two methods is that one accepts a parameter that specifies if the generation of the backward mapping is omitted or not. This can be useful if users want to declare both mapping directions themselves and therefore need no automatically generated backward mapping. The method without this parameter always generates both mappings and is considered the default. Both methods take an input stream to a mapping definition, the namespace URIs of the source and target ontologies of this mapping, and optionally an input stream to a JAR file containing additional property functions. The *XmlUtils* class is used to read the stream into a string that is then passed to the *QueryBuilder* class for query generation. After the *QueryBuilder* finished its work, the resulting queries are retrieved and added to a new *MappingElement* object, which then joins the other mappings with the source and target namespace URIs as identifier. After that, the newly created mapping is saved to disk by the *saveMapping* method and the storage is serialized to the directory file by the *serializeStorage* method. If a JAR file was submitted, it is extracted to the extension directory which is done with the *extractJar* method.

Besides the registering of new mappings, it is also possible to remove existing ones with the *removeMapping* method. The call of this method removes the mapping from the hash map in memory and deletes the mapping file on disk through the *deleteMapping* method.

6.3.1 Handlers

The classes in this package are responsible for decomposing a mapping definition into its different parts and transfer them into the proper query fragments. The handlers mirror the structure of the mapping definition format as for every mapping element exists a handler class. This means, there are handler classes for the namespace part, subject groups, and all three kinds of mappings. The handlers are also nested like in a definition file, that is the simple, nested, and complex mapping handlers are controlled by the subject group handler and only that and the namespace handler are managed by the *QueryBuilder* class from the mapping storage package. We use the XML Path Language (XPath)[2] for the actual segmentation of the individual parts. These are then passed to the respective handlers that process them again with the help of XPath expressions and create the equivalent query fragments. Figure 6.4 shows all classes and interfaces from this package.

[2] http://www.w3.org/TR/xpath20/

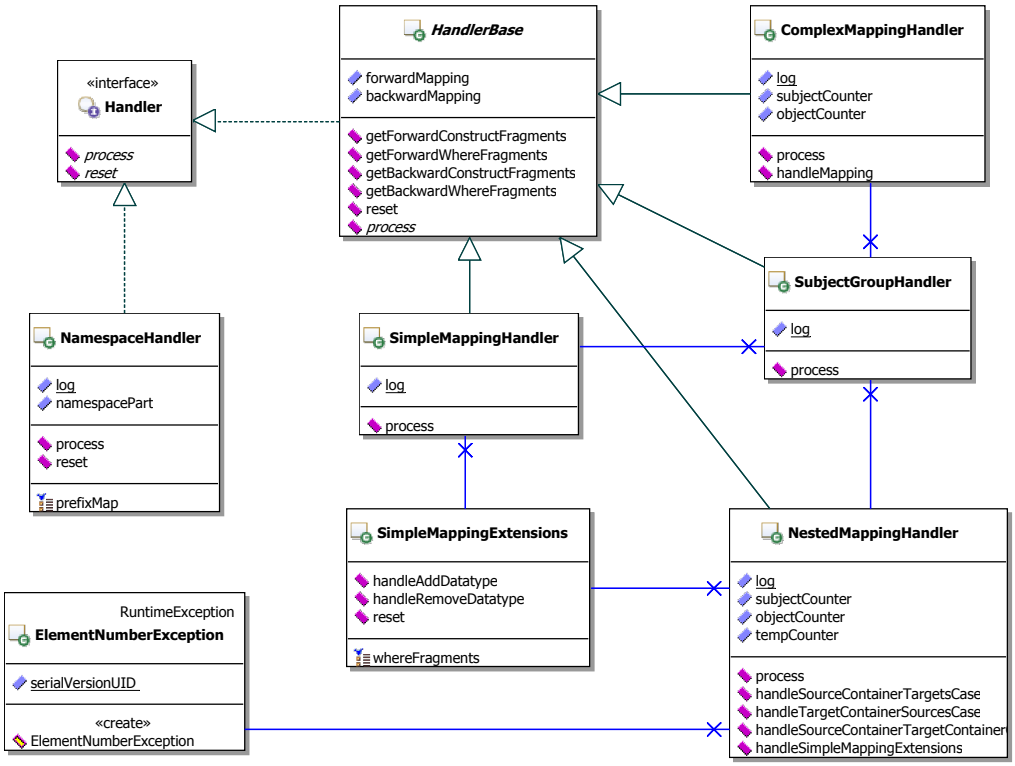

Figure 6.4: Class diagram of the Mapping Storage handlers

Handler

The *Handler* interface defines the basic operations that every handler class must support. Therefore, every handler class implements this interface directly or indirectly by extending a class that implements the interface. One of those basic operations is the *process* method that performs the analysis of the mapping definition part and then generates the adequate query fragments. The method takes a part of a mapping definition as a DOM Document and two integers. Both numbers are used for the unique naming of the variables in the generated triples of the query. The first is used for subject names and the second for objects. The other method is *reset*, it clears the internal state of a handler instance in order to make it reusable.

HandlerBase

HandlerBase is an abstract class that provides common functionality used by all handlers that process subject groups, simple, nested, or complex mappings. It contains two instances of the *Mapping* class from the mapping storage component, one for the forward and one for the backward mapping. These objects are used to cache the generated query fragments and therefore the *HandlerBase* class contains getter methods to retrieve those. In addition, this class provides an implementation for the *reset* method that simply clears the *Mapping* objects. Of course, the *process* method is not implemented as each mapping needs its own implementation and a default one would not make any sense.

NamespaceHandler

The *NamespaceHandler* class implements the *Handler* interface directly as it has no use for the functions defined in the *HandlerBase* class. It is used to process the namespace definitions in the mapping. As already stated in the introduction of this chapter, it uses XPath for the processing, more precisely it calls the *XmlUtil.queryDocument* method with the mapping document and the XPath expression *'/mappings/namespace'* to extract the namespace definitions. This results in a list of namespace nodes that are further split into the prefix and namespace URI part. These are finally used to create a query fragment representing a SPARQL namespace definition as shown in the example in Section 4.2.

SubjectGroupHandler

The *SubjectGroupHandler* class extends the *HandlerBase* class and adds its implementation of the *process* method. XPath expressions are used again to further dissect the input document. We remind the reader that every subject group must contain a source and target type and can contain simple, nested, and complex mappings. The types are treated by the *SubjectGroupHandler* itself, but the mappings are only separated and delegated to their respective handlers.

SimpleMappingHandler

As its name reveals, the *SimpleMappingHandler* class is responsible for processing simple mappings. Its *process* method is called from the *SubjectGroupHandler* and is only passed the part of the source document with one simple mapping at once. Consequently, the *SubjectGroupHandler* must call this method multiple times in a mapping with more than one `<simple-mapping>` element. However, this approach simplifies the task of the handler as it only needs to look for the `<source>` and `<target>` elements and transfer them into query fragments. A simple mapping generates two fragments if no extensions are used, one triple in the CONSTRUCT clause derived from the `<target>` element and one triple in the WHERE clause based on the `<source>` element. The triple in the WHERE part is enclosed in an OPTIONAL pattern as not every source document may contain it, but matches otherwise. If the simple mapping contains extensions, they are handled by the helper class *SimpleMappingExtensions* described in the next section.

SimpleMappingExtensions

The *SimpleMappingExtensions* is a helper class used to process the simple mapping extensions as described in Sections 4.3.2 and 4.3.6. It does not implement the *Handler* interface and is thus not a real handler class. It justifies its existence as a separate class because the simple mapping extensions can be used both in simple and nested mappings. Therefore, we implemented this class with two generic handling methods which can be used by the *SimpleMappingHandler* as well as the *NestedMappingHandler* class. The two methods are called *handleAddDatatype* and *handleRemoveDatatype*. They are practically identical with the sole difference that the first is used for adding a datatype with *fn:addDatatype* and the second variant is used for remove the datatype with the *fn:removeDatatype* property function as described in Section 4.3.2. The job of both methods lies in generating the query fragments of the WHERE clause that are needed for the extensions and each takes the same five arguments. The first three are strings called *datatype*, *datatypeConverter*, and *uriConverter*. They contain the values from the corresponding attributes of the `<source>` or `<target>` elements and can be 'null' if the respective attribute is not set, in which case they are ignored. The forth argument, *objectCounter*, is the same as in the *process* method of the handlers and is used to name the variables in the query. The fifth parameter is called *subjectName* and indicates the name of the subject variable used in the generated triples. This is due to the

generic nature of this class as the *SimpleMappingHandler* class produces top-level triples and the *NestedMappingHandler* nested ones that therefore use different names for their subjects. The generated query fragments are stored inside the class and can be retrieved by the caller with the *getWhereFragments* method.

There are three different simple mapping extensions: *datatype*, *datatypeConverter*, and *uriConverter*. Obviously, it is possible to use each one individually, but it is also possible to combine the *datatype* attribute with one of the two others. A combination of the two converter extensions or the use of all three would make no sense as every syntax conversion possible in a datatype converter can as well be done in an URI converter class.

NestedMappingHandler

Nested mappings are processed by the *NestedMappingHandler* class. It operates similar to the *SimpleMappingHandler* as it only receives the document part of one nested mapping at once. It analyzes the mapping to determine which of the three subcases introduced in Sections 4.3.3 to 4.3.5 it represents and calls the relevant method. Those methods operate all similar and generate query fragments for the involved containers and, like in a simple mapping, for the sources and targets. The exact details of those three cases can be found in the respective sections of Chapter 4. Nested mappings may also contain simple mapping extensions that are handled by the same helper class (*SimpleMappingExtensions*) as described in the prior section.

ComplexMappingHandler

Complex mappings can consist of two individual mappings, one for the forward and one for the backward direction. The *process* method of the *ComplexMappingHandler* first divides those and processes each one separately. As both of those mappings are structured identically, they are handled by just one method: *handleMapping*. This method first looks for `<args>` elements that get transferred into query fragments as shown in Section 4.3.7. In a second step, it uses the `<target>` element to build a normal triple for the CONSTRUCT clause. The `<source>` element normally contains a property function, but this results also in just a triple for the WHERE part of the query.

ElementNumberException

The *ElementNumberException* is a RuntimeException and is thrown if the number of `<source>` and `<target>` elements in a nested mapping do not match. This is a requirement that cannot be enforced by the XML Schema and therefore needs to be checked at runtime.

6.4 Remote Mapping Storage

The *Remote Mapping Storage* shares its functionality with the local *Mapping Storage*. They serve the same purpose in managing mappings and providing them on request. The difference is that the *Remote Mapping Storage* does not support generating new mappings from mapping definition files and that the remote version is an independent server process separated from the rest of the RDF Transformer. It is implemented with the Apache XML-RPC[3] library that provides an implementation of the XML-RPC protocol. Thereby, the communication between the RDF Transformer client and the Remote Mapping Storage server is based on remote procedure calls that use XML over HTTP. For more details, see the following sections and the Apache XML-RPC website.

[3] http://ws.apache.org/xmlrpc/

The *Remote Mapping Storage* is intended to be set up on a server connected to the Internet or the local network for serving its mappings to several RDF Transformer instances. The classes that compose the *Remote Mapping Storage* are presented in Figure 6.5.

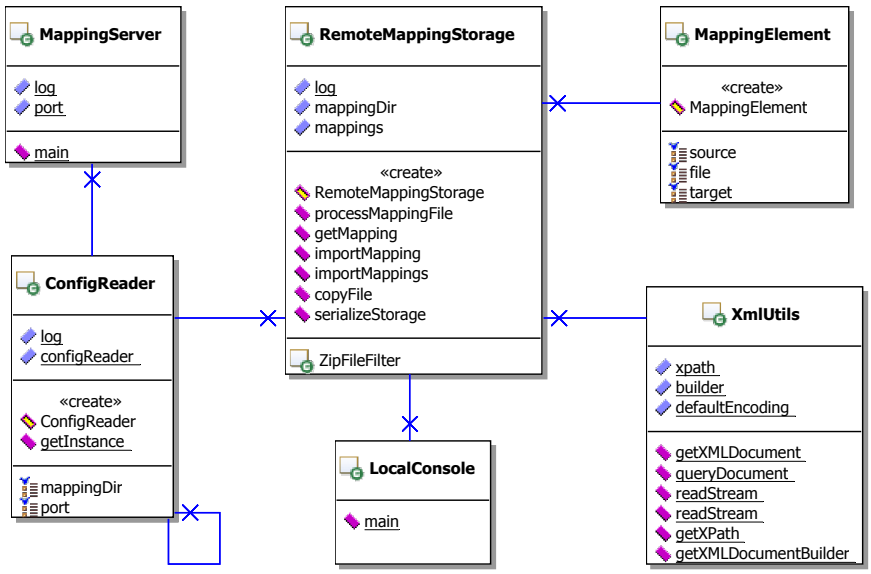

Figure 6.5: Class diagram of the Remote Mapping Storage component

LocalConsole

The *LocalConsole* is an interface for the local administration of a *Remote Mapping Storage* server. It can be used to import a single mapping available as a ZIP file or a complete directory with mappings and a directory file. These mappings are added to the already existent mappings at runtime and can be used instantly afterwards. Its command line syntax is explained in Appendix C.2.

ConfigReader

Due to the fact that the remote mapping storage is an independent application, it has its own configuration file and therefore needs its own implementation of the *ConfigReader* class to access these settings. This class is implemented like the corresponding class described in Section 6.2 with the difference that its configuration file is not an XML, but only a simple Java properties file called *'config.properties'*. An example of such a file with commented settings is presented in Appendix D.1.

MappingServer

The *MappingServer* class is the start class of the *Remote Mapping Storage* server. It first retrieves the port setting from the *ConfigReader* instance. After that, a new Apache XML-RPC web server is started that listens on this port and the request handlers are registered. Those are defined in the file *'Handler.properties'* whose format is defined by Apache and therefore not part of this work. Subsequently the *Remote Mapping Storage* server is ready to accept requests from RDF Transformers.

RemoteMappingStorage & MappingElement

The *RemoteMappingStorage* class contains a hash map with all known mappings. This map has the namespace URIs of the source and target ontologies as keys and instances of the class *MappingElement* as value elements. That class is not equal to the class with the same name of the local *Mapping Storage*. Indeed, they share their function as a data structure for mappings, but as the *Remote Mapping Storage* does only deliver mappings and not execute them, the contents of the class differ. This *MappingElement* class only contains the namespace URIs of the source and target ontology and the name of the ZIP file embodying the real mapping.

The *importMapping* and *importMappings* methods offer the possibility to import existing mappings from the local file system as already explained previously in this section. Furthermore, it is self-evident that this storage needs an equal mechanism to serialize the mapping directory if mappings are imported. This is provided by the *serializeStorage* method that is implemented like its counterpart in the local *MappingStorage* class. The main difference to that class lies in the *getMapping* method. Instead of returning an instance of the *MappingElement*, it returns the ZIP file containing the mapping. Thereto, the file is read from disk into a byte array in memory and send over the HTTP connection to the calling RDF Transformer, where it is locally registered and then applied. By means of this technique, the RDF Transformer receives the complete mapping and does not need to query the *Remote Mapping Storage* again for the same mapping.

XmlUtils

The *XmlUtils* class is a reduced version of the class with the same name from the RDF Transformer component. As the *Remote Mapping Storage* component does not process mapping definition files, all methods only used for that purpose were removed. For the implementation notes on the remaining methods consult Section 6.2.

6.5 Ontology Storage

The *Ontology Storage* component is responsible for the management of all ontology definitions that are used by the reasoner in the *ModelReader* class. Figure 6.6 shows that it consists of only two classes, the main class *OntologyStorage* and the supporting class *OntologyElement*. This class is comparable with the *MappingElement* class from the mapping storage package as it acts as the value element in the hash map that holds the ontologies and it is a data structure containing all the needed information about an ontology. In this case, this information is the ontology itself and the filename on disk where it is saved to.

The main class *OntologyStorage* is the equivalence of the *MappingStorage* class with the exception that ontologies cannot be generated so that it only stores them. The class is also implemented as a Singleton since only one instance must exist at any time. It supports three possibilities of adding ontologies to the storage, expressed as three overloaded methods called *registerOntology*. Those are: adding an ontology from an existing Jena model, reading it from a generic input stream, or letting it be downloaded automatically from the Web by its namespace URI. In all three cases, the ontology is saved to disk in a special location and a directory file is written there too. The mechanisms for those operations are equal to the ones used in the *MappingStorage* class and the reader is referred to that description in Section 6.3. Likewise, there exists methods to remove ontologies permanently (*removeOntology* & *deleteOntology*) and to retrieve existing ones (*getOntology*) that are implemented as their mapping storage counterparts.

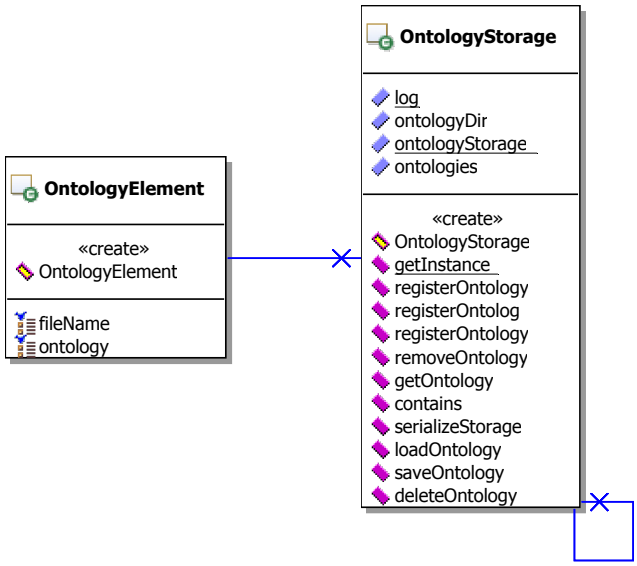

Figure 6.6: Class diagram of the Ontology Storage component

6.6 SPARQL Extensions

Figure 6.7 shows the standard set of extensions and ARQ property functions delivered with the RDF Transformer. The implementation of most classes is explained in Chapter 3 which leaves us here with the description of the overall interrelation and some notes on the implementation.

URI Converter Classes

The URI converter class hierarchy is depicted in the upper left corner of Figure 6.7. Every such class must implement the *IURIConverter* interface either directly or by extending the *URIConverterBase* base class. The interface defines the two methods used for converting between literals and URIs. The base class provides a default implementation for both of these methods that return the input unchanged. Based on this class or the interface, we can define a converter class for every imaginable URI scheme as shown with the examples of the *mailto*, *tel*, and *http* classes.

Datatype Converter Classes

In the upper right corner of Figure 6.7 stands the class hierarchy for the datatype converter classes. It is implemented similar to the URI converters with an interface defining a method for forward and backward transformation. This method performs the conversion between the two formats based on the form of the input. There is also a base class providing a default implementation that returns the input unchanged. Actual classes only need to implement the interface or extend the base class in order to be approved as datatype converter class.

ARQ Property Functions

The lower right corner of Figure 6.7 consists of a series of ARQ property functions that extend the *PFuncSimple* or *PFuncSimpleAndList* class provided by ARQ. They all override the *execEvaluated*

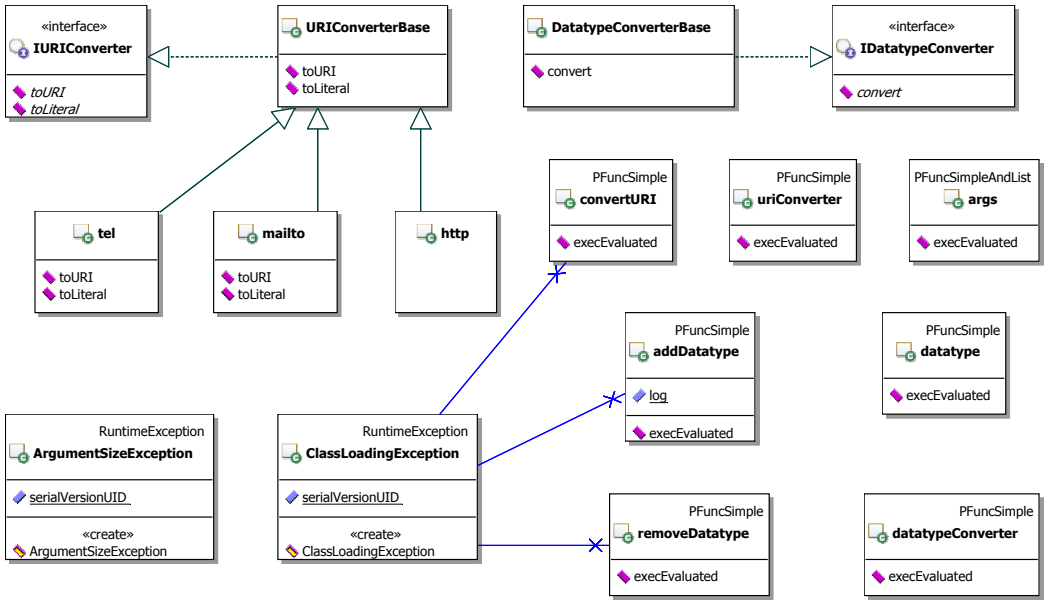

Figure 6.7: Class diagram of the SPARQL extension functions

method that gets called if a property function is used. As described in Chapter 3, these functions can store data in the ARQ execution context or create new objects based on their subjects with the use of the information in the execution context and other classes.

ClassLoadingException & ArgumentSizeException

Some property functions may need a specific number of arguments to work properly. If the actual number of arguments differs, the function is called in the wrong manner and an *ArgumentSizeException* is thrown to indicate this.

As a mapping can contain user defined converter classes, they must be loaded into the Java Virtual Machine. If those classes cannot be found on the classpath or cannot be loaded for any reason a *ClassLoadingException* is thrown to inform the user of this failure.

6.7 Configuration File & Security Policy

In this section, we address the possibilities of changing some settings of the RDF Transformer with the configuration file *'config.xml'* and how we deal with security concerns rising from the use of third-party code in the form of converter classes and custom ARQ property functions.

config.xml

The configuration file *'config.xml'* enables users to change some selected aspects of the RDF Transformer. The file has an XML syntax defined by the XML schema shown in Appendix B.4. An example *'config.xml'* is depicted in Listing 6.1.

```
<?xml version="1.0" encoding="UTF-8"?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
    xsi:noNamespaceSchemaLocation="schemas/config.xsd">
    <mappingDir>mappings</mappingDir>
    <ontologyDir>ontologies</ontologyDir>

    <remote-mapping-storages>
        <server>http://127.0.0.1:8090/xmlrpc</server>
    </remote-mapping-storages>

    <excludes>
        <ontology>http://webns.net/mvcb/</ontology>
    </excludes>
</config>
```

Listing 6.1: Example of a configuration file

With the configuration file, a user can change four different settings of the RDF Transformer. The first is the location of the mapping directory listed in the `<mappingDir>` XML element. If this option is changed the security policy file (described in the next section) and the start scripts must be adjusted accordingly as these files depend on the mappings location as well. Second, the directory where the ontologies are saved can be changed with the `<ontologyDir>` option. Enclosed in the `<remote-mapping-storages>` element is the third setting, the server addresses of the remote mapping storages. There can be any number of servers specified and every address is embedded in its own `<server>` subelement. Their arrangement in this file determines the order in which they are called later in the program. The last setting is listed under the `<excludes>` element and represents the ontologies that cannot or should not be downloaded automatically from their namespace URI by the *ModelReader*. If it encounters a namespace URI from this list, it knows the download is not desired or would fail and thus does not waste time in trying. This can happen if the ontology definition is not accessible by its namespace URI. The number of entries is also unrestricted and every URI is enclosed in its own `<ontology>` element.

The RDF Transformer contains hard coded default values for all these settings and therefore each one is optional. Even if the complete configuration file is missing, the application can run without any difficulty. But, if a configuration file is present the contained settings overwrite the default ones and only the missing settings remain at the hard coded values.

security.policy

The RDF Transformer is designed to receive mappings from different sources, with the inclusion of remote mapping storages even from the Internet. Thereby, we will face mappings that are defined by entities we do not know and cannot trust. As a mapping may contain executable code, we must ensure that this code causes no damage to our system and preserves our privacy. Therefore, we use the security technology provided by Java itself, namely the Java security manager with a corresponding security policy, to protect ourselves. Java provides developers with a fine grained set of permission that can be enabled based on different attributes of the code, among them the location of code called code base. Thus, we store all extension code from unknown sources in one directory and grant code originating from there no permissions at all. Listing 6.2 shows an example policy file respecting this rule.

```
// permissions for RDF Transformer code
grant codeBase "file:rdft.jar" {
    permission java.security.AllPermission;
```

```
};

// permissions for library code
grant codeBase "file:lib${/}-" {
    permission java.security.AllPermission;
};

// permissions for third-party extensions code
grant codeBase "file:mappings${/}ext${/}-" {
};
```

Listing 6.2: Example of a security policy file

The Java security model by default allocates no permission and as soon as we tell Java to use a security manager, we have to define the permission for all of our code. Hence, as can be seen in Listing 6.2 we cannot simply revoke certain rights from individual code files, but we have to grant the right permissions to every code entity that will be executed. More information about security policies and permissions in Java Standard Edition version 5 can be found in [Inc03] and [Inc02]. As code who runs without a security manager receives all permissions, we grant the same to our own code and the code of the libraries located in the '*lib*' subdirectory we use. Code in the directory where all additional classes are stored receives no permissions. This directory is the subdirectory '*ext*' in the mapping directory, therefore it becomes clear why a change of that location in the configuration file must be propagated. Normally started Java application do not use a security manager, which is why must start our application with the option '*-Djava.security.manager -Djava.security.policy=security.policy*' to enable the security manager and enforce the security policy.

Chapter 7

Evaluation

In this chapter, we want to present a full mapping between two real ontologies. On the one hand, this will show the reader a directly applicable mapping and how it is generated. On the other hand, it will also show the general practicability of our approach as we will use an ontology that was not regarded before in this work, not even in the requirements analysis in Chapter 3. The chapter is structured as follows: first, we introduce the two involved ontologies, then we present the example data expressed in both the source and target ontology. After that, we show the definition of the mapping that is used for the generation of the transformation queries which is shown in part four. At last, we look at how the just defined mapping is applied on the example data.

7.1 The Involved Ontologies

For the source ontology of our example, we choose the *SWRC* ontology already introduced in Section 3.1.4. Unlike in those sections, we focus this time not on the contact or event data but on the data about publications. This leads us to the target ontology used in this example, the *BibTeX* ontology. BibTeX itself is a popular and widespread plain text format for bibliographic data. With the *BibTeX* ontology, an attempt was made to create a corresponding Semantic Web format as described in [Kno04]. This ontology adheres closely to the original format and the respective part of *SWRC* orients itself at the plain text version of BibTeX as well. Hence, these two ontologies are ideal candidates for a mapping. There is one major difference between both ontologies and the plain text format. That is the order in properties describing authors, editors, and the like. In plain text BibTeX, each of them is represented as a single string that automatically contains and preserves the order of the individual entities. In both ontology formats, these concepts are implemented as single properties that are repeated for each entity, therefore no order can be assumed. This problem is not limited to only these ontologies but is a general problem of RDF and ontology definition as it appears often in situations like this.

7.2 Example Data

In this section, we present the example data that will be used in the evaluation. We show how the data is expressed in the source and the target ontology. The data is taken from the publications website of the AIFB institute at the University of Karlsruhe[1] and is publicly available. It shows

[1] http://www.aifb.uni-karlsruhe.de/Publikationen/showPublikationenSorted

the bibliographic data of a research article expressed in the *SWRC* ontology. For processing with the RDF Transformer, the relevant data about the authors was inlined in advance, which leads to the complete example data as shown in Listing 7.1. Although not stated in the *SWRC* ontology definition, many real world applications add datatypes to the *SWRC* properties as can be seen here too. Therefore, it would be legitimate to use them in our mapping definition as well.

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl:  <http://www.w3.org/2002/07/owl#> .
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#> .
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix swrc: <http://swrc.ontoware.org/ontology#> .

<http://www.aifb.uni-karlsruhe.de/Publikationen
    /viewPublikationOWL/id989.owl> rdf:type owl:Ontology ;
  rdfs:comment "Instance data for publication \"Transforming
                 Arbitrary Tables into F-Logic Frames with TARTAR\"" ;
  owl:imports <http://swrc.ontoware.org/ontology/portal> .

<http://www.aifb.uni-karlsruhe.de/Publikationen/viewPublikationOWL
    /id989instance> rdf:type swrc:Article ;
  swrc:abstract "The tremendous success
                 of the World Wide Web is countervailed by
                 (...)"^^xsd:string ;
  swrc:author <http://www.aifb.uni-karlsruhe.de/Personen/viewPersonOWL
               /id98instance> ,
              <http://www.aifb.uni-karlsruhe.de/Personen/viewPersonOWL
               /id57instance> ,
              <http://www.aifb.uni-karlsruhe.de/Personen/viewPersonOWL
               /id20instance> ,
              <http://www.aifb.uni-karlsruhe.de/Publikationen
               /viewExternerAutorOWL/id624instance> ,
              <http://www.aifb.uni-karlsruhe.de/Personen/viewPersonOWL
               /id2056instance> ,
              <http://www.aifb.uni-karlsruhe.de/Publikationen
               /viewExternerAutorOWL/id623instance> ;
  swrc:hasProject <http://www.aifb.uni-karlsruhe.de/Projekte
                   /viewProjektOWL/id50instance> ,
                  <http://www.aifb.uni-karlsruhe.de/Projekte
                   /viewProjektOWL/id35instance> ,
                  <http://www.aifb.uni-karlsruhe.de/Projekte
                   /viewProjektOWL/id42instance> ,
                  <http://www.aifb.uni-karlsruhe.de/Projekte
                   /viewProjektOWL/id49instance> ;
  swrc:isAbout <http://www.aifb.uni-karlsruhe.de/Forschungsgebiete
                /viewForschungsgebietOWL/id49instance> ,
               <http://www.aifb.uni-karlsruhe.de/Forschungsgebiete
```

```
                /viewForschungsgebietOWL/id79instance> ,
              <http://www.aifb.uni-karlsruhe.de/Forschungsgebiete
                /viewForschungsgebietOWL/id71instance> ,
              <http://www.aifb.uni-karlsruhe.de/Forschungsgebiete
                /viewForschungsgebietOWL/id81instance> ,
              <http://www.aifb.uni-karlsruhe.de/Forschungsgebiete
                /viewForschungsgebietOWL/id102instance> ,
              <http://www.aifb.uni-karlsruhe.de/Forschungsgebiete
                /viewForschungsgebietOWL/id137instance> ;
  swrc:journal "Data & Knowledge Engineering (DKE)"^^xsd:string ;
  swrc:number "3"^^xsd:string ;
  swrc:pages "567-595"^^xsd:string ;
  swrc:title "Transforming Arbitrary Tables into F-Logic Frames
              with TARTAR"^^xsd:string ;
  swrc:volume "60"^^xsd:string ;
  swrc:year "2007"^^xsd:string .

<http://www.aifb.uni-karlsruhe.de/Personen/viewPersonOWL/id20instance>
  rdf:type swrc:Person ;
  swrc:affiliation <http://www.aifb.uni-karlsruhe.de
                    /Forschungsgruppen/viewForschungsgruppeOWL
                    /id3instance> ;
  swrc:fax "+49 (721) 608 6580"^^xsd:string ;
  swrc:homepage "http://www.aifb.uni-karlsruhe.de/WBS/ysu"
                 ^^xsd:string ;
  swrc:name "York Sure"^^xsd:string ;
  swrc:phone "+49 (721) 608 6592"^^xsd:string ;
  swrc:photo "http://www.aifb.uni-karlsruhe.de/Personen/Bilder
              /U1p2l3o4a5d20"^^xsd:string .

<http://www.aifb.uni-karlsruhe.de/Forschungsgruppen
  /viewForschungsgruppeOWL/id3instance> rdf:type swrc:ResearchGroup .
<http://www.aifb.uni-karlsruhe.de/Forschungsgebiete
  /viewForschungsgebietOWL/id49instance> rdf:type swrc:ResearchTopic .
<http://www.aifb.uni-karlsruhe.de/Forschungsgebiete
  /viewForschungsgebietOWL/id71instance> rdf:type swrc:ResearchTopic .
<http://www.aifb.uni-karlsruhe.de/Forschungsgebiete
  /viewForschungsgebietOWL/id79instance> rdf:type swrc:ResearchTopic .
<http://www.aifb.uni-karlsruhe.de/Forschungsgebiete
  /viewForschungsgebietOWL/id81instance> rdf:type swrc:ResearchTopic .
<http://www.aifb.uni-karlsruhe.de/Forschungsgebiete
  /viewForschungsgebietOWL/id102instance> rdf:type swrc:ResearchTopic .
<http://www.aifb.uni-karlsruhe.de/Forschungsgebiete
  /viewForschungsgebietOWL/id137instance> rdf:type swrc:ResearchTopic .
```

```
<http://www.aifb.uni-karlsruhe.de/Projekte/viewProjektOWL
  /id35instance> rdf:type swrc:Project .
<http://www.aifb.uni-karlsruhe.de/Projekte/viewProjektOWL
  /id42instance> rdf:type swrc:Project .
<http://www.aifb.uni-karlsruhe.de/Projekte/viewProjektOWL
  /id49instance> rdf:type swrc:Project .
<http://www.aifb.uni-karlsruhe.de/Projekte/viewProjektOWL
  /id50instance> rdf:type swrc:Project .

<http://www.aifb.uni-karlsruhe.de/Publikationen/viewExternerAutorOWL
    /id623instance> rdf:type swrc:Person ;
  swrc:name "Matjaz Gams"^^xsd:string .

<http://www.aifb.uni-karlsruhe.de/Personen/viewPersonOWL/id98instance>
  rdf:type swrc:PhDStudent ;
  swrc:affiliation <http://www.aifb.uni-karlsruhe.de/Forschungsgruppen
                    /viewForschungsgruppeOWL/id3instance> ;
  swrc:fax "+49 (721) 608 6580"^^xsd:string ;
  swrc:name "Philipp Cimiano"^^xsd:string ;
  swrc:phone "+49 (721) 608 3705"^^xsd:string ;
  swrc:photo "http://www.aifb.uni-karlsruhe.de/WBS/pci/pci_bild.jpg"
              ^^xsd:string .

<http://www.aifb.uni-karlsruhe.de/Personen/viewPersonOWL/id57instance>
  rdf:type swrc:FullProfessor ;
  swrc:affiliation <http://www.aifb.uni-karlsruhe.de/Forschungsgruppen
                    /viewForschungsgruppeOWL/id3instance> ;
  swrc:fax "+49 (721) 608 6580"^^xsd:string ;
  swrc:name "Rudi Studer"^^xsd:string ;
  swrc:phone "+49 (721) 608 3923/4750"^^xsd:string ;
  swrc:photo "http://www.aifb.uni-karlsruhe.de/Personen/Bilder
              /U1p2l3o4a5d57"^^xsd:string .

<http://www.aifb.uni-karlsruhe.de/Publikationen/viewExternerAutorOWL
    /id624instance> rdf:type swrc:Person ;
  swrc:name "Vladislav Rajkovic"^^xsd:string .

<http://www.aifb.uni-karlsruhe.de/Personen/viewPersonOWL
    /id2056instance> rdf:type swrc:Person ;
  swrc:affiliation <http://www.aifb.uni-karlsruhe.de/Forschungsgruppen
                    /viewForschungsgruppeOWL/id3instance> ;
  swrc:fax ""^^xsd:string ;
  swrc:name "Aleksander Pivk"^^xsd:string ;
```

```
  swrc:phone ""^^xsd:string .
```

Listing 7.1: Example data in SWRC ontology

Listing 7.2 depicts the same data in the target ontology BibTeX. It was generated with the *SWRC* to *BibTeX* mapping defined in the next section. Unlike the data in the source vocabulary, we observe that the property names were substituted with the ones from the target ontology and each property has the right datatype attached. Furthermore, the nesting of the author names is broken up as their values appear in the respective properties.

```
@prefix rdfs:   <http://www.w3.org/2000/01/rdf-schema#> .
@prefix bibtex: <http://purl.oclc.org/NET/nknouf/ns/bibtex#> .
@prefix owl:    <http://www.w3.org/2002/07/owl#> .
@prefix xsd:    <http://www.w3.org/2001/XMLSchema#> .
@prefix rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix swrc:   <http://swrc.ontoware.org/ontology#> .

<http://www.aifb.uni-karlsruhe.de/Publikationen/viewPublikationOWL
    /id989instance> rdf:type bibtex:Article ;
  bibtex:hasAbstract "The tremendous success of the World Wide
                      Web is countervailed (...)"^^xsd:string ;
  bibtex:hasAuthor "Vladislav Rajkovic"^^xsd:string ,
                   "Rudi Studer"^^xsd:string ,
                   "Aleksander Pivk"^^xsd:string ,
                   "York Sure"^^xsd:string ,
                   "Matjaz Gams"^^xsd:string ,
                   "Philipp Cimiano"^^xsd:string ;
  bibtex:hasJournal "Data & Knowledge Engineering (DKE)"^^xsd:string ;
  bibtex:hasNumber "3"^^xsd:string ;
  bibtex:hasPages "567-595"^^xsd:string ;
  bibtex:hasTitle "Transforming Arbitrary Tables into F-Logic Frames
                   with TARTAR"^^xsd:string ;
  bibtex:hasVolume "60"^^xsd:nonNegativeInteger ;
  bibtex:hasYear "2007"^^xsd:nonNegativeInteger .
```

Listing 7.2: Example data in BibTeX ontology

7.3 Mapping Definition

As both the source and the target ontology adhere closely to the plain text format, a lot of mappings are simple ones that only substitute the namespace parts and the local names. Before we introduce the mapping definition, we first must mention another detail of the original BibTeX format and the implementation in the two ontologies we use in this example. Plain text BibTeX uses classes that describe the kind of bibliographic object it collects information about and each of those can have a series of properties. The classes themselves range from articles over books to unpublished work, with altogether over a dozen of such classes. Each can have more or less the same set of properties like *author*, the *year* in that it was published, how many *pages* it has, and so

on. In both our studied ontologies those classes are mapped to OWL classes and the properties to OWL properties.

In Listing 7.3, we see the mapping definition for the *Article* class used to generate the transformations from *SWRC* to *BibTeX* and back. Due to the peculiarities of BibTeX described in the prior paragraph and as this mapping definition is already quite long, we renounce to repeat basically the same definition for every BibTeX class. Let us just say that for a complete mapping from *SWRC* to *BibTeX*, we would need to replicate the mapping of the properties for every class in its own subject group. Nevertheless, the mapping in Listing 7.3 is fully functional and can at least be used to transform data about SWRC Articles as demonstrated in this chapter.

```
<?xml version="1.0" encoding="UTF-8"?>
<mappings xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="schemas/mapping.xsd">
  <namespace prefix="swrc">
    http://swrc.ontoware.org/ontology#
  </namespace>
  <namespace prefix="bibtex">
    http://purl.oclc.org/NET/nknouf/ns/bibtex#
  </namespace>
  <namespace prefix="xsd">
    http://www.w3.org/2001/XMLSchema#
  </namespace>

  <subject-group>
    <source-type>swrc:Article</source-type>
    <target-type>bibtex:Article</target-type>

    <!-- simple mappings with datatype extension -->
    <simple-mapping>
      <source>swrc:abstract</source>
      <target datatype="xsd:string">bibtex:hasAbstract</target>
    </simple-mapping>
    <simple-mapping>
      <source>swrc:address</source>
      <target datatype="xsd:string">bibtex:hasAddress</target>
    </simple-mapping>
    <simple-mapping>
      <source>swrc:booktitle</source>
      <target datatype="xsd:string">bibtex:hasBooktitle</target>
    </simple-mapping>
    <simple-mapping>
      <source>swrc:chapter</source>
      <target datatype="xsd:nonNegativeInteger">
        bibtex:hasChapter
      </target>
    </simple-mapping>
```

```
<simple-mapping>
  <source>swrc:edition</source>
  <target datatype="xsd:string">bibtex:hasEdition</target>
</simple-mapping>
<simple-mapping>
  <source>swrc:isbn</source>
  <target datatype="xsd:string">bibtex:hasISBN</target>
</simple-mapping>
<simple-mapping>
  <source>swrc:journal</source>
  <target datatype="xsd:string">bibtex:hasJournal</target>
</simple-mapping>
<simple-mapping>
  <source>swrc:keywords</source>
  <target datatype="xsd:string">bibtex:hasKeywords</target>
</simple-mapping>
<simple-mapping>
  <source>swrc:location</source>
  <target datatype="xsd:string">bibtex:hasLocation</target>
</simple-mapping>
<simple-mapping>
  <source>swrc:month</source>
  <target datatype="xsd:string">bibtex:hasMonth</target>
</simple-mapping>
<simple-mapping>
  <source>swrc:note</source>
  <target datatype="xsd:string">bibtex:hasNote</target>
</simple-mapping>
<simple-mapping>
  <source>swrc:number</source>
  <target datatype="xsd:string">bibtex:hasNumber</target>
</simple-mapping>
<simple-mapping>
  <source>swrc:pages</source>
  <target datatype="xsd:string">bibtex:hasPages</target>
</simple-mapping>
<simple-mapping>
  <source>swrc:price</source>
  <target datatype="xsd:string">bibtex:hasPrice</target>
</simple-mapping>
<simple-mapping>
  <source>swrc:series</source>
  <target datatype="xsd:string">bibtex:hasSeries</target>
</simple-mapping>
  <simple-mapping>
```

```
<source>swrc:title</source>
  <target datatype="xsd:string">bibtex:hasTitle</target>
</simple-mapping>
<simple-mapping>
  <source>swrc:type</source>
  <target datatype="xsd:string">bibtex:hasType</target>
</simple-mapping>
<simple-mapping>
  <source>swrc:volume</source>
  <target datatype="xsd:nonNegativeInteger">
    bibtex:hasVolume
  </target>
</simple-mapping>
<simple-mapping>
  <source>swrc:year</source>
  <target datatype="xsd:nonNegativeInteger">
    bibtex:hasYear
  </target>
</simple-mapping>

<!-- nested mappings with datatype extension -->
<nested-mapping>
  <target datatype="xsd:string">bibtex:hasAffiliation</target>
  <source-container name="swrc:affiliation">
    <source>swrc:name</source>
  </source-container>
</nested-mapping>
<nested-mapping>
  <target datatype="xsd:string">bibtex:hasAuthor</target>
  <source-container name="swrc:author">
    <source>swrc:name</source>
  </source-container>
</nested-mapping>
<nested-mapping>
  <target datatype="xsd:string">bibtex:hasEditor</target>
  <source-container name="swrc:editor">
    <source>swrc:name</source>
  </source-container>
</nested-mapping>
<nested-mapping>
  <target datatype="xsd:string">bibtex:hasInstitution</target>
  <source-container name="swrc:institution">
    <source>swrc:name</source>
  </source-container>
</nested-mapping>
```

```
    <nested-mapping>
      <target datatype="xsd:string">bibtex:hasOrganization</target>
      <source-container name="swrc:organization">
        <source>swrc:name</source>
      </source-container>
    </nested-mapping>
    <nested-mapping>
      <target datatype="xsd:string">bibtex:hasPublisher</target>
      <source-container name="swrc:publisher">
        <source>swrc:name</source>
      </source-container>
    </nested-mapping>
    <nested-mapping>
      <target datatype="xsd:string">bibtex:hasSchool</target>
      <source-container name="swrc:school">
        <source>swrc:name</source>
      </source-container>
    </nested-mapping>

  </subject-group>
</mappings>
```

Listing 7.3: SWRC to BibTeX ontology mapping definition

After the analysis of the two ontologies, we can identify three main differences. First, both ontologies use basically the same names for the properties with the small difference that the *BibTeX* ontology prefixes all with 'has'. This poses no problem as this kind of translation is achieved with each of our mappings, but it makes our job of finding them easier. Second, *SWRC* does not define typed properties, but the *BibTeX* ontology does. Therefore, we need to specify the *datatype* attribute in each `<target>` element to add a datatype in the transformation. Mappings embodying this difference are arranged as the group of simple mappings in Listing 7.3. The third and most problematic difference is that some properties of the *SWRC* ontology use other *SWRC* classes as their range, whereas *BibTeX* uses plain strings. An example for that is the property representing the author of a work. In *SWRC* this is *author* that takes an object as its value. Mostly, this object will be an instance of the *Person* class also from *SWRC*. Otherwise, the *hasAuthor* property from the *BibTeX* ontology uses a literal as value that contains only the name of the author as a string. To overcome this discrepancy, we can use a nested mapping as the *swrc:Person* class contains a *name* property with the equivalent content for the *BibTeX* author property. Likewise, we can find such properties for all the similar cases as shown in the group of nested mappings in Listing 7.3. Please note, that with the current implementation of the RDF Transformer, this solution only works if the additional data of the nested classes is stored in the same RDF resource. If it is only linked, it cannot be found as the Transformer does not automatically reload linked data. We think, the download of additional data lies in the responsibility of the superordinate application as it is impossible for a generic library as the RDF Transformer to know which links should be followed and how deep. An application that uses our component should therefore first aggregate all needed data before it passes all of it to the RDF Transformer.

7.4 Transformation Query

After a mapping definition is developed, the RDF Transformer uses it to create the forward and backward transformation queries. The one for the forward mapping is depicted in Listing 7.4. We can see that except the *rdf:type* triple, every triple is enclosed in an OPTIONAL to enable the matching of resources that do not contain all properties. In addition, we see how the datatype is added for every mapping and towards the end of the query how nested mappings are handled.

```
PREFIX rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX fn:     <java:ch.uzh.ifi.rdftransformer.sparqlext.>
PREFIX swrc:   <http://swrc.ontoware.org/ontology#>
PREFIX bibtex: <http://purl.oclc.org/NET/nknouf/ns/bibtex#>
PREFIX xsd:    <http://www.w3.org/2001/XMLSchema#>

CONSTRUCT {
  ?subject1 rdf:type               bibtex:Article .
  ?subject1 bibtex:hasAbstract     ?object1 .
  ?subject1 bibtex:hasAddress      ?object2 .
  ?subject1 bibtex:hasBooktitle    ?object3 .
  ?subject1 bibtex:hasChapter      ?object4 .
  ?subject1 bibtex:hasEdition      ?object5 .
  ?subject1 bibtex:hasISBN         ?object6 .
  ?subject1 bibtex:hasJournal      ?object7 .
  ?subject1 bibtex:hasKeywords     ?object8 .
  ?subject1 bibtex:hasLocation     ?object9 .
  ?subject1 bibtex:hasMonth        ?object10 .
  ?subject1 bibtex:hasNote         ?object11 .
  ?subject1 bibtex:hasNumber       ?object12 .
  ?subject1 bibtex:hasPages        ?object13 .
  ?subject1 bibtex:hasPrice        ?object14 .
  ?subject1 bibtex:hasSeries       ?object15 .
  ?subject1 bibtex:hasTitle        ?object16 .
  ?subject1 bibtex:hasType         ?object17 .
  ?subject1 bibtex:hasVolume       ?object18 .
  ?subject1 bibtex:hasYear         ?object19 .
  ?subject1 bibtex:hasAffiliation  ?object20 .
  ?subject1 bibtex:hasAuthor       ?object23 .
  ?subject1 bibtex:hasEditor       ?object26 .
  ?subject1 bibtex:hasInstitution  ?object29 .
  ?subject1 bibtex:hasOrganization ?object32 .
  ?subject1 bibtex:hasPublisher    ?object35 .
  ?subject1 bibtex:hasSchool       ?object38 .
}
WHERE {
  ?subject1 rdf:type swrc:Article .
  OPTIONAL { ?subject1 swrc:abstract ?temp1 .
```

```
  ?temp1 fn:datatype    xsd:string .
  ?temp1 fn:addDatatype ?object1 }
OPTIONAL { ?subject1 swrc:address ?temp2 .
  ?temp2 fn:datatype    xsd:string .
  ?temp2 fn:addDatatype ?object2 }
OPTIONAL { ?subject1 swrc:booktitle ?temp3 .
  ?temp3 fn:datatype    xsd:string .
  ?temp3 fn:addDatatype ?object3 }
OPTIONAL { ?subject1 swrc:chapter ?temp4 .
  ?temp4 fn:datatype    xsd:nonNegativeInteger .
  ?temp4 fn:addDatatype ?object4 }
OPTIONAL { ?subject1 swrc:edition ?temp5 .
  ?temp5 fn:datatype    xsd:string .
  ?temp5 fn:addDatatype ?object5 }
OPTIONAL { ?subject1 swrc:isbn ?temp6 .
  ?temp6 fn:datatype    xsd:string .
  ?temp6 fn:addDatatype ?object6 }
OPTIONAL { ?subject1 swrc:journal ?temp7 .
  ?temp7 fn:datatype    xsd:string .
  ?temp7 fn:addDatatype ?object7 }
OPTIONAL { ?subject1 swrc:keywords ?temp8 .
  ?temp8 fn:datatype    xsd:string .
  ?temp8 fn:addDatatype ?object8 }
OPTIONAL { ?subject1 swrc:location ?temp9 .
  ?temp9 fn:datatype    xsd:string .
  ?temp9 fn:addDatatype ?object9 }
OPTIONAL { ?subject1 swrc:month ?temp10 .
  ?temp10 fn:datatype    xsd:string .
  ?temp10 fn:addDatatype ?object10 }
OPTIONAL { ?subject1 swrc:note ?temp11 .
  ?temp11 fn:datatype    xsd:string .
  ?temp11 fn:addDatatype ?object11 }
OPTIONAL { ?subject1 swrc:number ?temp12 .
  ?temp12 fn:datatype    xsd:string .
  ?temp12 fn:addDatatype ?object12 }
OPTIONAL { ?subject1 swrc:pages ?temp13 .
  ?temp13 fn:datatype    xsd:string .
  ?temp13 fn:addDatatype ?object13 }
OPTIONAL { ?subject1 swrc:price ?temp14 .
  ?temp14 fn:datatype    xsd:string .
  ?temp14 fn:addDatatype ?object14 }
OPTIONAL { ?subject1 swrc:series ?temp15 .
  ?temp15 fn:datatype    xsd:string .
  ?temp15 fn:addDatatype ?object15 }
OPTIONAL { ?subject1 swrc:title ?temp16 .
```

```
  ?temp16 fn:datatype    xsd:string .
  ?temp16 fn:addDatatype ?object16 }
OPTIONAL { ?subject1 swrc:type ?temp17 .
  ?temp17 fn:datatype    xsd:string .
  ?temp17 fn:addDatatype ?object17 }
OPTIONAL { ?subject1 swrc:volume ?temp18 .
  ?temp18 fn:datatype    xsd:nonNegativeInteger .
  ?temp18 fn:addDatatype ?object18 }
OPTIONAL { ?subject1 swrc:year ?temp19 .
  ?temp19 fn:datatype    xsd:nonNegativeInteger .
  ?temp19 fn:addDatatype ?object19 }
OPTIONAL { ?subject1 swrc:affiliation ?temp20 .
OPTIONAL { ?temp20 swrc:name ?innertemp20 .
  ?innertemp20 fn:datatype    xsd:string .
  ?innertemp20 fn:addDatatype ?object20 }
}
OPTIONAL { ?subject1 swrc:author ?temp23 .
OPTIONAL { ?temp23 swrc:name ?innertemp23 .
  ?innertemp23 fn:datatype    xsd:string .
  ?innertemp23 fn:addDatatype ?object23 }
}
OPTIONAL { ?subject1 swrc:editor ?temp26 .
OPTIONAL { ?temp26 swrc:name ?innertemp26 .
  ?innertemp26 fn:datatype    xsd:string .
  ?innertemp26 fn:addDatatype ?object26 }
}
OPTIONAL { ?subject1 swrc:institution ?temp29 .
OPTIONAL { ?temp29 swrc:name ?innertemp29 .
  ?innertemp29 fn:datatype    xsd:string .
  ?innertemp29 fn:addDatatype ?object29 }
}
OPTIONAL { ?subject1 swrc:organization ?temp32 .
OPTIONAL { ?temp32 swrc:name ?innertemp32 .
  ?innertemp32 fn:datatype    xsd:string .
  ?innertemp32 fn:addDatatype ?object32 }
}
OPTIONAL { ?subject1 swrc:publisher ?temp35 .
OPTIONAL { ?temp35 swrc:name ?innertemp35 .
  ?innertemp35 fn:datatype    xsd:string .
  ?innertemp35 fn:addDatatype ?object35 }
}
OPTIONAL { ?subject1 swrc:school ?temp38 .
OPTIONAL { ?temp38 swrc:name ?innertemp38 .
  ?innertemp38 fn:datatype    xsd:string .
  ?innertemp38 fn:addDatatype ?object38 }
```

```
  }
}
```

Listing 7.4: SWRC to BibTeX forward mapping query

Listing 7.5 shows the automatically generated backward query for this mapping. The WHERE clause shows that every datatype is removed again by the *fn:removeDatatype* property function and in the CONSTRUCT clause, we see how the nested mappings translate into the creation of blank nodes containing the *swrc:name* property.

```
PREFIX rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX fn:     <java:ch.uzh.ifi.rdftransformer.sparqlext.>
PREFIX swrc:   <http://swrc.ontoware.org/ontology#>
PREFIX bibtex: <http://purl.oclc.org/NET/nknouf/ns/bibtex#>
PREFIX xsd:    <http://www.w3.org/2001/XMLSchema#>

CONSTRUCT {
  ?subject1  rdf:type          swrc:Article .
  ?subject1  swrc:abstract     ?object1 .
  ?subject1  swrc:address      ?object2 .
  ?subject1  swrc:booktitle    ?object3 .
  ?subject1  swrc:chapter      ?object4 .
  ?subject1  swrc:edition      ?object5 .
  ?subject1  swrc:isbn         ?object6 .
  ?subject1  swrc:journal      ?object7 .
  ?subject1  swrc:keywords     ?object8 .
  ?subject1  swrc:location     ?object9 .
  ?subject1  swrc:month        ?object10 .
  ?subject1  swrc:note         ?object11 .
  ?subject1  swrc:number       ?object12 .
  ?subject1  swrc:pages        ?object13 .
  ?subject1  swrc:price        ?object14 .
  ?subject1  swrc:series       ?object15 .
  ?subject1  swrc:title        ?object16 .
  ?subject1  swrc:type         ?object17 .
  ?subject1  swrc:volume       ?object18 .
  ?subject1  swrc:year         ?object19 .
  ?subject1  swrc:affiliation  _:temp20 .
    _:temp20 swrc:name         ?object21 .
  ?subject1  swrc:author       _:temp23 .
    _:temp23 swrc:name         ?object24 .
  ?subject1  swrc:editor       _:temp26 .
    _:temp26 swrc:name         ?object27 .
  ?subject1  swrc:institution  _:temp29 .
    _:temp29 swrc:name         ?object30 .
  ?subject1  swrc:organization _:temp32 .
    _:temp32 swrc:name         ?object33 .
```

```
  ?subject1  swrc:publisher    _:temp35 .
    _:temp35 swrc:name         ?object36 .
  ?subject1  swrc:school       _:temp38 .
    _:temp38 swrc:name         ?object39 .
}
WHERE {
  ?subject1 rdf:type bibtex:Article .
  OPTIONAL { ?subject1 bibtex:hasAbstract  ?temp1 .
    ?temp1  fn:removeDatatype ?object1 }
  OPTIONAL { ?subject1 bibtex:hasAddress   ?temp2 .
    ?temp2  fn:removeDatatype ?object2 }
  OPTIONAL { ?subject1 bibtex:hasBooktitle ?temp3 .
    ?temp3  fn:removeDatatype ?object3 }
  OPTIONAL { ?subject1 bibtex:hasChapter   ?temp4 .
    ?temp4  fn:removeDatatype ?object4 }
  OPTIONAL { ?subject1 bibtex:hasEdition   ?temp5 .
    ?temp5  fn:removeDatatype ?object5 }
  OPTIONAL { ?subject1 bibtex:hasISBN      ?temp6 .
    ?temp6  fn:removeDatatype ?object6 }
  OPTIONAL { ?subject1 bibtex:hasJournal   ?temp7 .
    ?temp7  fn:removeDatatype ?object7 }
  OPTIONAL { ?subject1 bibtex:hasKeywords  ?temp8 .
    ?temp8  fn:removeDatatype ?object8 }
  OPTIONAL { ?subject1 bibtex:hasLocation  ?temp9 .
    ?temp9  fn:removeDatatype ?object9 }
  OPTIONAL { ?subject1 bibtex:hasMonth     ?temp10 .
    ?temp10 fn:removeDatatype ?object10 }
  OPTIONAL { ?subject1 bibtex:hasNote      ?temp11 .
    ?temp11 fn:removeDatatype ?object11 }
  OPTIONAL { ?subject1 bibtex:hasNumber    ?temp12 .
    ?temp12 fn:removeDatatype ?object12 }
  OPTIONAL { ?subject1 bibtex:hasPages     ?temp13 .
    ?temp13 fn:removeDatatype ?object13 }
  OPTIONAL { ?subject1 bibtex:hasPrice     ?temp14 .
    ?temp14 fn:removeDatatype ?object14 }
  OPTIONAL { ?subject1 bibtex:hasSeries    ?temp15 .
    ?temp15 fn:removeDatatype ?object15 }
  OPTIONAL { ?subject1 bibtex:hasTitle     ?temp16 .
    ?temp16 fn:removeDatatype ?object16 }
  OPTIONAL { ?subject1 bibtex:hasType      ?temp17 .
    ?temp17 fn:removeDatatype ?object17 }
  OPTIONAL { ?subject1 bibtex:hasVolume    ?temp18 .
    ?temp18 fn:removeDatatype ?object18 }
  OPTIONAL { ?subject1 bibtex:hasYear      ?temp19 .
    ?temp19 fn:removeDatatype ?object19 }
```

```
  OPTIONAL { ?subject1 bibtex:hasAffiliation  ?innertemp21 .
    ?innertemp21 fn:removeDatatype ?object21 }
  OPTIONAL { ?subject1 bibtex:hasAuthor       ?innertemp24 .
    ?innertemp24 fn:removeDatatype ?object24 }
  OPTIONAL { ?subject1 bibtex:hasEditor       ?innertemp27 .
    ?innertemp27 fn:removeDatatype ?object27 }
  OPTIONAL { ?subject1 bibtex:hasInstitution  ?innertemp30 .
    ?innertemp30 fn:removeDatatype ?object30 }
  OPTIONAL { ?subject1 bibtex:hasOrganization ?innertemp33 .
    ?innertemp33 fn:removeDatatype ?object33 }
  OPTIONAL { ?subject1 bibtex:hasPublisher    ?innertemp36 .
    ?innertemp36 fn:removeDatatype ?object36 }
  OPTIONAL { ?subject1 bibtex:hasSchool       ?innertemp39 .
    ?innertemp39 fn:removeDatatype ?object39 }
}
```

Listing 7.5: SWRC to BibTeX backward mapping query

If we had created the complete mapping covering all classes from BibTeX, we would get such a query pair for each class or rather each subject group.

This mapping was only an example, it is not meant to be the one and only *SWRC* to *BibTeX* mapping. Other users may decide that the generation of blank nodes in the backward query is not satisfying. As an alternative, those users could instruct the RDF Transformer to only generate the forward query and then define a *BibTeX* to *SWRC* mapping with different content for the backward query generation. Furthermore, complex queries, that would conserve more information about the nesting object, could be used as well.

7.5 Application of the Mapping

After we have defined the mapping and the RDF Transformer has generated transformation queries based on that definition, we can use the Transformer to apply it to data. As already mentioned in the presentation of the example data in Section 7.2, we used the RDF Transformer to create the *BibTeX* ontology version of this data shown in Listing 7.2. For that, we took the *SWRC* data in Listing 7.1 as the input data and the mapping depicted in Listing 7.3. Note that the data contains not all elements from the mapping (for instance there are no *editor*, *month*, or *publisher* properties), but thanks to the OPTIONAL keyword used in the transformation queries the mapping still works. Furthermore, there are properties in the example data (like *isAbout*) which are not present in the mapping as there are no corresponding properties in the *BibTeX* ontology. This information is lost in the transformation and will not be available in a backward mapping.

Based on this generated BibTeX data, we apply the backward query to transform our example data back into its original *SWRC* vocabulary. The result of this operation can be seen in Listing 7.6. As just explained, the data does not match the original source data completely as some information was not transferred in the first mapping. Another noteworthy fact is that there are numerous properties referring to empty blank nodes. Those are caused by the way the transformation queries are implemented. They can appear for elements defined in the mapping but not occurring in the data. They do not falsify the result and should not cause any trouble in subsequent processing.

```
@prefix dc:      <http://purl.org/dc/elements/1.1/> .
```

```
@prefix foaf:    <http://xmlns.com/foaf/0.1/> .
@prefix units:   <http://zeitkunst.org/fontomri/0.01/units.owl#> .
@prefix dcterms: <http://purl.org/dc/terms/> .
@prefix wot:     <http://xmlns.com/wot/0.1/> .
@prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#> .
@prefix bibtex:  <http://purl.oclc.org/NET/nknouf/ns/bibtex#> .
@prefix dctype:  <http://purl.org/dc/dcmitype/> .
@prefix owl:     <http://www.w3.org/2002/07/owl#> .
@prefix xsd:     <http://www.w3.org/2001/XMLSchema#> .
@prefix rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix swrc:    <http://swrc.ontoware.org/ontology#> .

<http://www.aifb.uni-karlsruhe.de/Publikationen/viewPublikationOWL
    /id989instance> rdf:type swrc:Article ;
  swrc:abstract "The tremendous success of the World Wide
                 Web is countervailed (...)" ;
  swrc:title "Transforming Arbitrary Tables into F-Logic Frames
              with TARTAR" ;
  swrc:volume "60" ;
  swrc:year "2007" ;
  swrc:journal "Data & Knowledge Engineering (DKE)" ;
  swrc:number "3" ;
  swrc:pages "567-595" ;

  swrc:author [ swrc:name "Rudi Studer" ] ;
  swrc:author [ swrc:name "York Sure" ] ;
  swrc:author [ swrc:name "Vladislav Rajkovic" ] ;
  swrc:author [ swrc:name "Matjaz Gams" ] ;
  swrc:author [ swrc:name "Aleksander Pivk" ] ;
  swrc:author [ swrc:name "Philipp Cimiano" ] ;

  swrc:affiliation [ ] ;
  swrc:affiliation [ ] ;
  swrc:affiliation [ ] ;
  swrc:affiliation [ ] ;
  swrc:affiliation [ ] ;
  swrc:affiliation [ ] ;

  swrc:editor [ ] ;
  swrc:editor [ ] ;
  swrc:editor [ ] ;
  swrc:editor [ ] ;
  swrc:editor [ ] ;
  swrc:editor [ ] ;
```

```
  swrc:institution [ ] ;
  swrc:institution [ ] ;
  swrc:institution [ ] ;
  swrc:institution [ ] ;
  swrc:institution [ ] ;
  swrc:institution [ ] ;

  swrc:organization [ ] ;
  swrc:organization [ ] ;
  swrc:organization [ ] ;
  swrc:organization [ ] ;
  swrc:organization [ ] ;
  swrc:organization [ ] ;

  swrc:publisher [ ] ;
  swrc:publisher [ ] ;
  swrc:publisher [ ] ;
  swrc:publisher [ ] ;
  swrc:publisher [ ] ;
  swrc:publisher [ ] ;

  swrc:school [ ] ;
  swrc:school [ ] ;
  swrc:school [ ] ;
  swrc:school [ ] ;
  swrc:school [ ] ;
  swrc:school [ ] .
```

Listing 7.6: Example BibTeX data transformed back to SWRC

In conclusion, we have shown in this chapter that our mapping language can be applied to ontologies different from the ones analyzed in Chapter 3 as well. Furthermore, we presented how the definition of a mapping is done and how the RDF Transformer generates the transformation queries from it. Those queries were depicted as well and used to transform example data to a target ontology and back again.

Chapter 8

Example Application: Semantic Clipboard

In order to further demonstrate the usefulness of the RDF Transformer, we embedded it into a directly usable example application. This application is called *Semantic Clipboard* and is the topic of this chapter. The general idea of this application and the scripts used for the copy and paste (on MacOS X) operations are adapted from [Lau07]. The Semantic Clipboard has the same elementary functions like a normal clipboard as it allows data to be copied from one application to another. The big difference is that the data is semantic data expressed in RDF and that the Semantic Clipboard can transform it into different ontologies, of course by using the RDF Transformer. Figure 8.1 shows a schematic overview of the Semantic Clipboard.

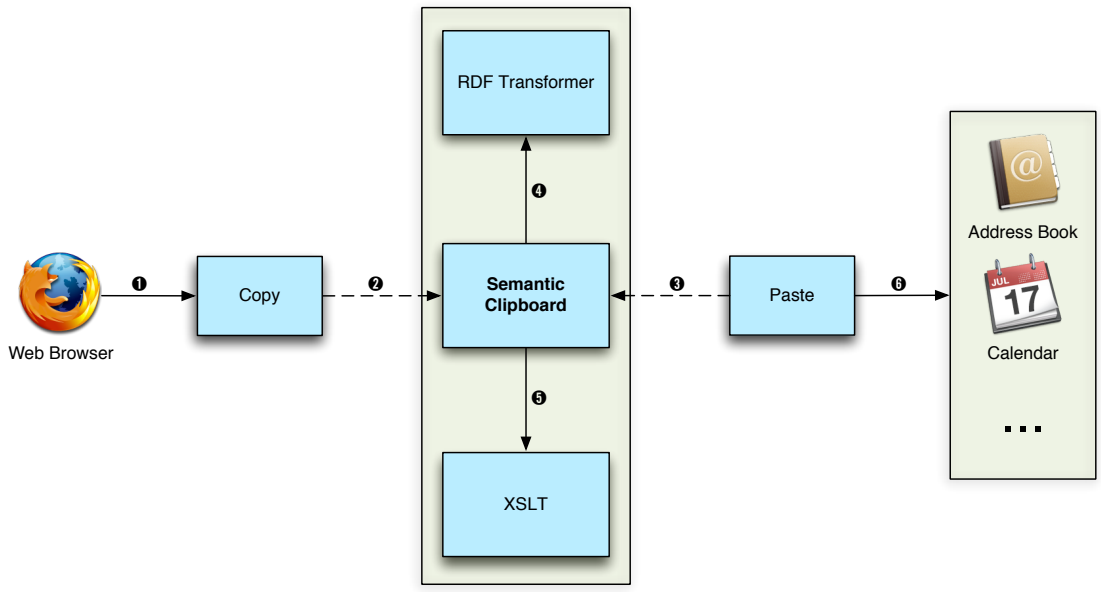

Figure 8.1: Overview of the Semantic Clipboard

As every other clipboard, the Semantic Clipboard supports two operations to interact with it. The first is the *copy* operation that is used to read the source data from the input application into

the clipboard. The second operation is *paste* that writes the stored data to the output application. The main distinction to a normal clipboard lies in the transformation of the data in-between with a call to the RDF Transformer. The application that issues a paste passes the namespace of the target ontology with the request. The Semantic Clipboard then converts the already stored input data with the help of the RDF Transformer and returns the translated data to the calling application.

There are two possibilities to integrate this clipboard into an existing system. The first would be to modify all source and target applications, so that they use the Semantic Clipboard directly. This would be a great deal of work and not even possible at all times, particularly in closed source systems. Thus, we implemented the second alternative that provides tools at clearly defined extension points of several applications. In this way, the Semantic Clipboard may not be deeply integrated into the operating system which makes its use slightly different, but it becomes easily portable across multiple systems. In our implementation, we achieve the *copy* mechanism through a bookmarklet in a browser that enables us to copy semantic data from Web pages. The extension of other application would be possible as well, but we focus in this example solely on Web pages as they are an important part of the Semantic Web. On the other side is the *paste* mechanism that is implemented as a script associated with a system wide keyboard shortcut. The script identifies the current foreground application and uses this information to determine the target format. Our implementation comes preconfigured with support for the address book, the calendar, and a BibTeX management software, but it is easy to add other applications. With the information of the target format, the script makes a *paste* request to the Semantic Clipboard and then writes the result data to a temporary file on disk. At last, the system is instructed to open that file, which brings the data into the application. Although, this is not the most elegant way of integration, it shows nicely how the RDF Transformer can be used in a real world application.

The interface of the Semantic Clipboard is realized via the HTTP protocol. After startup, the clipboard acts as a HTTP server and listens on a given port for incoming connections. A *copy* operation is initiated from the Web browser with a click on the copy bookmarklet (step one in Figure 8.1). This bookmarklet loads additional JavaScript code from a web server which analyzes the source Web pages for links to RDF data. These URLs are extracted and each is send separately by a HTTP request to the Semantic Clipboard (marked as step two in Figure 8.1). The clipboard takes this URL from the request string and stores it internally until a *paste* request is received or it is overwritten with data from a new *copy* operation. That completes the copy request and the Semantic Clipboard waits again for new connections. Now a *paste* request can be send (step three) that must contain an identifier for the desired target format. This can either be the namespace URI of an ontology or an identifying URI, if the target format should be plain text. Unfortunately, the support for RDF data in popular application is still limited, which is why we need to transform the RDF data into their plain text counterpart for those application. This is done with Extensible Stylesheet Language Transformations (XSLT)[1] and specific stylesheets defined for the ontologies that have a plain text equivalent. After the clipboard received the *paste* request, the source data URL is feed into the RDF Transformer that downloads the data for processing and the target URI is extracted from the request string. As this URI can refer to either an ontology or a plain text format, the clipboard consults a mapping table that contains all known plain text formats and maps them to corresponding ontology namespace URIs. This mapping table is constructed from the file '*ontologies.map*' that contains matching pairs of plain text identifiers and ontology namespaces. If the plain text format has a corresponding ontology, this operation yields an URI suitable as target parameter for the RDF Transformer. If an RDF format was requested, we get such an URI directly and need no intermediate mapping step. The RDF Transformer now has all information it needs and can execute the translation of the input data, which is depicted as step four in Figure 8.1. The resulting data, expressed in the target vocabulary, is then cached in the Semantic Clipboard until a new transformation is executed. Depending on the requested output

[1] http://www.w3.org/TR/xslt20/

format, this data is directly returned as a HTTP response to the caller (step six) or the plain text to ontology mapping must first be reversed. As already mentioned, this is done with XSLT (in step five) that generates the plain text form from the RDF data with the help of stylesheets. Those are selected from a second mapping table that is created from the file *'stylesheets.map'* which contains pairs of identifying URLs and the names of the appropriate stylesheet files in the *'stylesheets'* subdirectory of the application directory. The return of the transformed data marks the end of this paste request. It is now possible to retrieve the same data again with an identical request, transform the original data into a different format with another *paste* request containing a new target URI, or start over with a new call to the *copy* operation.

Chapter 9

Conclusion

The Semantic Web provides great prospects to describe the semantics of data with languages like RDF and OWL. As a secondary effect, this creates the problem that there are multiple ontologies developed that relate to the same or overlapping domains. Thereby arises the need for a transformation service that enables different application to exchange data even if they do not use the same ontology.

In this thesis, we introduced such a transformation service. First, we investigated in Chapter 3 how ontologies covering at least parts of the same domain can differ in representing that information. This yielded in a set of requirements for a mapping language which was presented in Chapter 4. Our approach was to define a mapping in a simple XML format and use SPARQL for the actual transformation. The CONSTRUCT queries in SPARQL, as a powerful mechanism for such transformations of RDF data, turned out to be ideal for this purpose. That way, we were able to reuse existing Semantic Web technology and could benefit from its power and functionality. The big challenge then lay in the definition of the mapping format and the generation of the transformation queries from those files. Based on the findings from Chapter 3, we defined three different kinds of mappings as presented in Chapter 4 as well. Each of these mappings received its own handler class responsible for transferring mapping definitions in transformation queries. Those handler classes and the rest of the prototype implementation were described in Chapter 6. In Chapter 7, we demonstrated the usefulness of our approach and the prototype in a longer example with real world ontologies and data. In order to achieve its full potential, a transformation service like this needs to be embedded in another application which was done exemplary in the form of a simple Semantic Clipboard in Chapter 8.

The major contribution of our thesis is that we can map more information between more diverse ontologies. Our approach is more powerful than former attempts of ontology mapping that tried to declare equivalence between individual properties. For instance, the use of *owl:equivalentClass* in combination with a reasoner enables only direct mappings from one isolated element onto another. This corresponds to our simple one to one mappings introduced in Section 3.3.1, but as shown in the same section, even our simple mappings provide more flexibility with datatype modification, typing, and conversions between URIs and literals. Beyond that, our approach offers mappings involving nesting of elements, both on the source and target side of a transformation and almost arbitrary processing possibilities with complex mappings. They enable, for example, the restoring of implicit knowledge that would otherwise be lost in a transformation as explained in Section 3.3.7. Admittedly, this flexibility comes at the cost of additional programming effort on the part of the user, but even without utilizing this functionality, our mapping language is very powerful and applicable to a broad number of mapping cases. In addition, it is in most cases possible to generate usable mappings in both directions from only one mapping definition. This can cut the effort to define a round-trip mapping in half as only one

mapping must be defined.

Future work could be done in integrating the transformation service in different applications and operating systems. In the case of extensible, open systems it could even be implemented so that the average user does not notice any difference in usage and experiences only the advantages it brings. Furthermore, domain experts will need to define more mappings and distribute them, for example via a publicly available Remote Mapping Storage server on the Internet, so that mappings between many ontologies become easily possible.

Appendix A

Additional Java Source Code

A.1 *datatypeConverter* Java Source Code

The class shown in Listing A.1 defines an ARQ property function that can be used in the conversion of untyped to typed literals described in Section 3.3.2. Its use is optional and only needed if the syntax of the source and target ontology do not match.

```
package ch.uzh.ifi.rdftransformer.sparqlext;

import com.hp.hpl.jena.graph.Node;
import com.hp.hpl.jena.sparql.engine.ExecutionContext;
import com.hp.hpl.jena.sparql.engine.QueryIterator;
import com.hp.hpl.jena.sparql.engine.binding.Binding;
import com.hp.hpl.jena.sparql.pfunction.PFLib;
import com.hp.hpl.jena.sparql.pfunction.PFuncSimple;
import com.hp.hpl.jena.sparql.util.Symbol;

/**
 * This class is an ARQ property function. It stores the fully
 * qualified name of the Java converter class used for syntax
 * conversions by the addDatatype and removeDatatype property
 * functions in the ARQ execution context.
 * @author Matthias Hert
 *
 */
public class datatypeConverter extends PFuncSimple {

    @Override
    public QueryIterator execEvaluated(Binding binding, Node subject,
            Node predicate, Node object, ExecutionContext execCxt) {
        execCxt.getContext().set(Symbol.create("datatypeConverter"),
                object.toString(false));
```

```
        return PFLib.result(binding, execCxt);
    }
}
```

Listing A.1: Java source code for property function *datatypeConverter*

A.2 *removeDatatype* Java Source Code

The class shown in Listing A.2 defines an ARQ property function that is used in the backwards mapping of the untyped to typed literal case. For that purpose, it removes the datatype of a typed literal and therefore transforms it into an untyped one.

```
package ch.uzh.ifi.rdftransformer.sparqlext;

import ch.uzh.ifi.rdftransformer.sparqlext.IDatatypeConverter;

import com.hp.hpl.jena.graph.Node;
import com.hp.hpl.jena.sparql.core.Var;
import com.hp.hpl.jena.sparql.engine.ExecutionContext;
import com.hp.hpl.jena.sparql.engine.QueryIterator;
import com.hp.hpl.jena.sparql.engine.binding.Binding;
import com.hp.hpl.jena.sparql.pfunction.PFLib;
import com.hp.hpl.jena.sparql.pfunction.PFuncSimple;
import com.hp.hpl.jena.sparql.util.Symbol;

/**
 * This class is an ARQ property function. It is used to remove
 * the datatype of a typed literal and optionally to convert
 * the syntax of the input.
 * @author Matthias Hert
 *
 */
public class removeDatatype extends PFuncSimple {

    @Override
    public QueryIterator execEvaluated(Binding binding, Node subject,
            Node predicate, Node object, ExecutionContext execCxt) {
        // get name of optional datatype converter class
        String converter = execCxt.getContext().getAsString(Symbol
                .create("datatypeConverter"));
        // remove entry from execution context for further calls
        execCxt.getContext().remove(
                Symbol.create("datatypeConverter"));
        String resultValue = null;
```

```
        // perform syntax conversion if requested
        if (converter != null) {
            IDatatypeConverter conv = null;
            try {
                conv = (IDatatypeConverter)Class.forName(converter)
                    .newInstance();
            }
            catch (IllegalAccessException ex) {
                throw new ClassLoadingException(
                        "Could not access converter class!");
            }
            catch (InstantiationException ex) {
                throw new ClassLoadingException(
                        "Could not instantiate converter class!");
            }
            catch (ClassNotFoundException ex) {
                throw new ClassLoadingException(
                        "Converter class not found!");
            }
            resultValue = conv.convert(
                    subject.getLiteralLexicalForm());
        }
        // apply without conversion
        else {
            resultValue = subject.getLiteralLexicalForm();
        }

        // create untyped result
        Node result = Node.createLiteral(resultValue,
                subject.getLiteralLanguage(), false);
        return PFLib.oneResult(binding, Var.alloc(object),
                result, execCxt);
    }
}
```

Listing A.2: Java source code for property function *removeDatatype*

Appendix B

XML Schemata

B.1 XML Schema Definition of Mapping Language

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified">
  <xs:element name="mappings">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" ref="namespace"/>
        <xs:element maxOccurs="unbounded" ref="subject-group"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="namespace">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:anyURI">
          <xs:attribute name="prefix" use="required" type="xs:NCName"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="subject-group">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="1"  maxOccurs="unbounded"
            ref="source-type"/>
        <xs:element minOccurs="1"  maxOccurs="unbounded"
            ref="target-type"/>
        <xs:choice maxOccurs="unbounded">
          <xs:element ref="nested-mapping"/>
```

```
        <xs:element ref="simple-mapping"/>
        <xs:element ref="complex-mapping"/>
      </xs:choice>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="source-type" type="xs:NMTOKEN"/>
<xs:element name="target-type" type="xs:NMTOKEN"/>
<xs:element name="nested-mapping">
  <xs:complexType>
    <xs:choice>
        <xs:sequence>
          <xs:element maxOccurs="unbounded" ref="target"/>
          <xs:element ref="source-container"/>
        </xs:sequence>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" ref="source"/>
        <xs:element ref="target-container"/>
      </xs:sequence>
      <xs:sequence>
        <xs:element ref="source-container"/>
        <xs:element ref="target-container"/>
      </xs:sequence>
    </xs:choice>
  </xs:complexType>
</xs:element>
<xs:element name="source-container">
  <xs:complexType>
    <xs:sequence>
      <xs:element maxOccurs="unbounded" ref="source"/>
    </xs:sequence>
    <xs:attribute name="name" use="required" type="xs:NMTOKEN"/>
    <xs:attribute name="type" type="xs:NMTOKEN"/>
  </xs:complexType>
</xs:element>
<xs:element name="target-container">
  <xs:complexType>
    <xs:sequence>
      <xs:element maxOccurs="unbounded" ref="target"/>
    </xs:sequence>
    <xs:attribute name="name" use="required" type="xs:NMTOKEN"/>
    <xs:attribute name="type" type="xs:NMTOKEN"/>
  </xs:complexType>
</xs:element>
<xs:element name="simple-mapping">
```

```
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="source"/>
        <xs:element ref="target"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="complex-mapping">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="forward-mapping"/>
        <xs:element minOccurs="0" ref="backward-mapping"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="forward-mapping">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="arg"/>
        <xs:element ref="source"/>
        <xs:element ref="target"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="backward-mapping">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="arg"/>
        <xs:element ref="source"/>
        <xs:element ref="target"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="source">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:NMTOKEN">
          <xs:attribute name="datatype" type="xs:anyURI"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="target">
    <xs:complexType>
      <xs:simpleContent>
```

```
        <xs:extension base="xs:NMTOKEN">
          <xs:attribute name="datatype" type="xs:anyURI"/>
          <xs:attribute name="datatypeConverter" type="xs:NCName"/>
          <xs:attribute name="uriConverter" type="xs:NCName"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="arg" type="xs:NMTOKEN"/>
</xs:schema>
```

Listing B.1: XML Schema definition of mapping language

B.2 XML Schema Definition of Mapping Directory File

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified">
  <xs:element name="mappings">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="mapping"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="mapping">
    <xs:complexType>
      <xs:attribute name="file" use="required" type="xs:NMTOKEN"/>
      <xs:attribute name="source" use="required" type="xs:anyURI"/>
      <xs:attribute name="target" use="required" type="xs:anyURI"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Listing B.2: XML Schema definition of mapping directory file

B.3 XML Schema Definition of Ontology Directory File

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified">
  <xs:element name="ontologies">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" maxOccurs="unbounded"
          ref="ontology"/>
```

```
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="ontology">
    <xs:complexType>
      <xs:attribute name="file" use="required" type="xs:NMTOKEN"/>
      <xs:attribute name="name" use="required" type="xs:anyURI"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Listing B.3: XML Schema definition of ontology directory file

B.4 XML Schema Definition of Configuration File

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified">
  <xs:element name="config">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" ref="mappingDir"/>
        <xs:element minOccurs="0" ref="ontologyDir"/>
        <xs:element minOccurs="0" ref="remote-mapping-storages"/>
        <xs:element minOccurs="0" ref="excludes"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="mappingDir" type="xs:anyURI"/>
  <xs:element name="ontologyDir" type="xs:anyURI"/>
  <xs:element name="remote-mapping-storages">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="server"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="server" type="xs:anyURI"/>
  <xs:element name="excludes">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" maxOccurs="unbounded"
          ref="ontology"/>
      </xs:sequence>
    </xs:complexType>
```

```
  </xs:element>
  <xs:element name="ontology" type="xs:anyURI"/>
</xs:schema>
```

Listing B.4: XML Schema definition of configuration file

Appendix C

Command Line Syntaxes

C.1 RDF Transformer: TestConsole

The *TestConsole* class acts as a local command line interface to the RDF Transformer as described in Section 6.2. It supports a number of arguments that are explained in Table C.1.

Option	Description
--mapping	This option is used for registering new mappings. It specifies the file containing the mapping definition.
--jar	This option is used for registering new mappings. It specifies the file containing the custom property functions.
--source	This option is used for registering new mappings. It specifies the namespace URI of the source ontology.
--target	This option is used for registering new mappings as well as for transforming data. It specifies the namespace URI of the target ontology.
--ontology	This option is used for registering a new ontology. It specifies its namespace URI.
--ontology-file	This option is used for the local registration of a new ontology. It specifies the file containing the ontology definition.
--data	This option is used for transforming data. It specifies the file containing the input data.
--base	This option is used for transforming data. It specifies the base URI used for converting relative to absolute URIs in the input data.
--in-syntax	This option is used for transforming data. It specifies the syntax of the input data. If omitted, the default syntax RDF/XML is used.
--out-syntax	This option is used for transforming data. It specifies the syntax of the transformed data. If omitted, the default syntax RDF/XML is used.
--result	This option is used for transforming data. It specifies the file where the transformed output data is written to.

Table C.1: Arguments of the RDF Transformer TestConsole class

Each option must be followed directly with its value only separated through a blank. The arguments can be submitted in any order as long as all needed options for a specific operation are present. The *TestConsole* supports the following three operations:

Registering a new mapping In order to register a new mapping, the options *--mapping*, *--source*,

and *--target* must be set.

Registering a new ontology A new ontology can be registered in two ways. Either, we use only the *--ontology* argument and the RDF Transformer tries to download the ontology from the namespace URI or we specify the *--ontology-file* option too, in which case the ontology definition is read from that local file and not from the Internet.

Transforming data The main operation is transforming data which needs at least the *--target* and *--data* arguments. If only those two are given, the input is assumed to be in the RDF/XML syntax and the transformed output is written to the command line in the same syntax. Therefore, it is optionally possible to specify the syntax of the input with *--in-syntax* and the one of the output with *--out-syntax* as well as a filename for the output with *--result*. Additionally, the base URI used for converting relative URIs to absolute ones in the input data can be specified with the *--base* option. If it is omitted and the data contains relative URIs, it will not be possible to write it in the RDF/XML syntax.

The *TestConsole* supports the combination of those operations as well. All that is needed for this is setting the necessary arguments and every operation is executed in the order as listed above.

C.2 Remote Mapping Storage: LocalConsole

The *LocalConsole* class acts as a local command line interface to the Remote Mapping Storage as described in Section 6.4. It supports a number of arguments that are explained in Table C.2.

Option	Description
--source	This option is used for importing one new mapping. It specifies the namespace URI of the source ontology.
--target	This option is used for importing one new mapping. It specifies the namespace URI of the target ontology.
--file	This option is used for importing one new mapping. It specifies the file containing the mapping.
--directory	This option is used for importing a whole mapping directory. It specifies the directory containing the mappings and the mapping directory file.

Table C.2: Arguments of the Remote Mapping Storage LocalConsole class

Each option must be followed directly with its value only separated through a blank. The arguments can be submitted in any order as long as all needed options for a specific operation are present. The *LocalConsole* supports the following two operations:

Importing one new mapping To import one specific mapping we need to provide the *--source*, *--target*, and *--file* options.

Importing a whole mapping directory For the import of a whole mapping directory we only need to set the *--directory* option.

Those two options are exclusive and therefore cannot be combined.

Appendix D

Configuration Files

D.1 MappingServer Configuration File

The configuration file used by the *MappingServer* class of the Remote Mapping Storage is a simple Java property file. An example file with comments is shown in Listing D.1.

```
# directory where the mappings are stored
mappingDir=mappings

# port
port=8090
```

Listing D.1: Example MappingServer configuration file

It can contain the two entries *'mappingDir'* to specify a different mapping directory and *'port'* to change the port on which the server listens for incoming connections.

References

[AvH04] Grigoris Antoniou and Frank van Harmelen. *A Semantic Web Primer*. The MIT Press, 2004.

[BBG00] Massimo Benerecetti, Paolo Bouquet, and Chiara Ghidini. Contextual reasoning distilled. *Journal of Theoretical and Experimental Artificial Intelligence*, 12(3):279–305, 2000.

[BHLT06] Tim Brady, Dave Hollander, Andrew Layman, and Richard Tobin. Namespaces in xml 1.0 (second edition). `http://www.w3.org/TR/xml-names/`, August 2006. Last visited February 2008.

[BLFM05] Tim Berners-Lee, Roy Fielding, and Larry Masinter. Uniform resource identifier (uri): Generic syntax. `http://tools.ietf.org/html/rfc3986`, January 2005. Last visited February 2008.

[BM07] Dan Brickley and Libby Miller. Foaf vocabulary specification 0.91. `http://xmlns.com/foaf/spec/`, November 2007. Last visited February 2008.

[Bou05] Paolo Bouquet. D2.2.1v2: Specification of a common framework for characterizing alignment. `http://knowledgeweb.semanticweb.org/semanticportal/deliverables/D2.2.1v2.pdf`, 2005. Last visited February 2008.

[BPSM+06] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and Francois Yergeau. Extensible markup language (xml) 1.0 (fourth edition). `http://www.w3.org/TR/xml/`, September 2006. Last visited February 2008.

[Bri06] Dan Brickley. Basic geo (wgs84 lat/long) vocabulary. `http://www.w3.org/2003/01/geo/`, February 2006. Last visited February 2008.

[CM05] Dan Connolly and Libby Miller. Rdf calendar - an application of the resource description framework to icalendar data. `http://www.w3.org/TR/rdfcal/`, September 2005. Last visited February 2008.

[DH98] Frank Dawson and Tim Howes. vcard mime directory profile. `http://www.ietf.org/rfc/rfc2426.txt`, September 1998. Last visited February 2008.

[Doa02] AnHai Doan. *Learning to Map between Structured Representations of Data*. PhD thesis, University of Washington, 2002.

[DS98] Frank Dawson and Derik Stenerson. Internet calendaring and scheduling core object specification (icalendar). `http://www.ietf.org/rfc/rfc2445.txt`, November 1998. Last visited February 2008.

[Euz04] Jérôme Euzenat. D2.2.3: State of the art on ontology alignment. http://knowledgeweb.semanticweb.org/semanticportal/deliverables/D2.2.3.pdf, 2004. Last visited February 2008.

[Gru93] Thomas R. Gruber. A translation approach to portable ontology specifications. http://ksl.stanford.edu/knowledge-sharing/papers/ontolingua-intro.rtf, April 1993. Last visited February 2008.

[HSW06] Harry Halpin, Brian Suda, and Norman Walsh. An ontology for vcards. http://www.w3.org/2006/vcard/ns, November 2006. Last visited February 2008.

[Ian01] Renato Iannella. Representing vcard objects in rdf/xml. http://www.w3.org/TR/vcard-rdf, February 2001. Last visited February 2008.

[Inc02] Sun Microsystems Inc. Permissions in the java 2 standard edition development kit (jdk). http://java.sun.com/j2se/1.5.0/docs/guide/security/permissions.html, 2002. Last visited February 2008.

[Inc03] Sun Microsystems Inc. Default policy implementation and policy file syntax. http://java.sun.com/j2se/1.5.0/docs/guide/security/PolicyFiles.html, September 2003. Last visited February 2008.

[Jen] Jena - a semantic web framework for java. http://jena.sourceforge.net. Last visited February 2008.

[KBS07] Christoph Kiefer, Abraham Bernstein, and Markus Stoker. The fundamentals of isparql - a virtual triple approach for similarity-based semantic web tasks. In *Proceedings of the 6th International Semantic Web Conference (ISWC)*, Lecture Notes in Computer Science, pages 295–309. Springer, 2007.

[KC04] Graham Klyne and Jeremy J. Carroll. Resource description framework (rdf): Concepts and abstract syntax. http://www.w3.org/TR/rdf-concepts/, February 2004. Last visited February 2008.

[Kno04] Nick Knouf. bibtex definition in web ontology language (owl) version 0.1. http://zeitkunst.org/bibtex/0.1/, 2004. Last visited February 2008.

[Kru] Sebastian Ryszard Kruk. Rdftranslator. http://wiki.corrib.org/index.php/RDFTranslator. Last visited February 2008.

[KS03] Yannis Kalfoglou and Marco Schorlemmer. If-map: An ontology-mapping method based on information-flow theory. *The Knowledge Engineering Review*, 18(1):1–31, 2003.

[KVV06] Markus Krtzsch, Denny Vrandecic, and Max Vlkel. Semantic mediawiki. In *Proceedings of the 5th International Semantic Web Conference (ISWC06)*, volume 4273 of *Lecture Notes in Computer Science*, pages 935–942. Springer, November 2006.

[Lau07] Gian Marco Laube. Semclip - ontology mediation and content negotiation for the semantic clipboard. Master's thesis, University of Zurich, 2007.

[Mar] Marcont initiative. http://www.marcont.org. Last visited February 2008.

[MBS08] Alistair Miles, Thomas Baker, and Ralph Swick. Best practice recipes for publishing rdf vocabularies. http://www.w3.org/TR/swbp-vocab-pub/, January 2008. Last visited February 2008.

[MM04] Frank Manola and Eric Miller. Rdf primer. http://www.w3.org/TR/rdf-primer/, February 2004. Last visited February 2008.

[NM01] Natalya Noy and Mark Musen. Anchor-prompt: Using non-local context for semantic matching. In *IJCAI 2001 workshop on ontology and information sharing*, pages 63–70, 2001.

[OW02] Thomas Ottmann and Peter Widmayer. *Algorithmen und Datenstrukturen*. Spektrum Akademischer Verlag, 2002.

[Pro] Protégé - the ontology editor and knowledge acquisition system. http://protege.stanford.edu. Last visited February 2008.

[SBH+05] York Sure, Stephan Bloehdorn, Peter Haase, Jens Hartmann, and Daniel Oberle. The swrc ontology - semantic web for research communities. In *Proceedings of the 12th Portuguese Conference on Artificial Intelligence - Progress in Artificial Intelligence (EPIA 2005)*, volume 3803 of *LNCS*, pages 218–231, 2005.

[SH07] Andy Seaborne and Matthias Hert. Calculate total time from variable number of single times in a property function. http://tech.groups.yahoo.com/group/jena-dev/message/31211, October 2007. Last visited February 2008.

[SMT00] C. M. Sperberg-McQueen and Henry Thompson. Xml schema. http://www.w3.org/XML/Schema, April 2000. Last visited February 2008.

[SMW08] Semantic mediawiki - help:annotation. http://semantic-mediawiki.org/index.php/Help:Annotation, January 2008. Last visited February 2008.

[Uni07] What is unicode? http://www.unicode.org/standard/WhatIsUnicode.html, 2007. Last visited February 2008.

[Wal05a] Norman Walsh. Extracting vcards from hcard markup. http://norman.walsh.name/2005/12/12/vcard, December 2005. Last visited February 2008.

[Wal05b] Norman Walsh. Modelling vcards in rdf. http://norman.walsh.name/2005/12/05/vcard, December 2005. Last visited February 2008.