SPARQL Rewriting for Query Mediation over Mapped Ontologies

by

Konstantinos E. Makris

A thesis submitted to the Department of Electronic & Computer Engineering of the Technical University of Crete in partial fulfilment of the requirements for the Diploma of Electronic Engineer and Computer Engineer

advisor: Prof. Stavros Christodoulakis

Chania, July 2010
Dedication

To my mom and dad.

Thank you for all your love and support.
Abstract

In the recent years establishing interoperability and supporting data integration has become a major research challenge for the web of data. Uniform information access of heterogeneous sources is of major importance for Semantic Web applications and end users. We describe a methodology for SPARQL query mediation over federated OWL/RDF knowledge bases. The query mediation exploits mappings between semantically related entities of the mediator ontology (global ontology) and the federated site ontologies (local ontologies). A very rich set of mapping types, based on Description Logic semantics, is supported. The SPARQL queries which are posed over the global ontology are decomposed, rewritten, and then submitted to the federated sources. The rewritten SPARQL queries are locally evaluated and the results are returned to the mediator. We describe the formal modeling of executable mappings (i.e. mappings that can be used in SPARQL query rewriting), as well as the theoretic and implementation aspects of SPARQL query rewriting. Finally, we describe the implementation of a system supporting the mediation process.
Acknowledgements

I would like to thank my supervisor, Prof. Stavros Christodoulakis, for his encouragement and his continuous guidance and support throughout my research. I would like also to thank him for the important experiences he offered me during my stay at the Laboratory of Distributed Multimedia Information Systems and Applications (MUSIC).

I would like to express my gratitude to the readers of this thesis Mr. Antonios Deligiannakis and Mr. Michail G. Lagoudakis for the time they devoted and their critical evaluation.

My appreciation goes to Nektarios Gioldasis for his supervision and his valuable help regarding this thesis. I am also grateful to Nikos Bikakis for being always ready to offer his help whenever needed.

I would like to thank Chrisa Tsinaraki for the time she devoted mostly in the beginning of my research, as well as the rest staff of the Laboratory for the pleasant environment they provided.

Finally, I wish to thank my brothers Zisis and Dimitris, as well as Eleni and Panagiots for helping me get through the difficult times, and for all the emotional support, entertainment, and caring they provided.

Konstantinos E. Makris
Technical University of Crete
July 2010
Contents

List of Tables vii

List of Figures viii

1 Introduction 1

2 Background 3
  2.1 Resource Description Framework and Schema (RDF/S) . . . . . . . . . . . . 3
  2.2 Web Ontology Language (OWL) ........................................ 7
  2.3 SPARQL query language ................................................ 11
    2.3.1 SPARQL syntax .................................................... 11
    2.3.2 Graph patterns ................................................... 13
    2.3.3 Query forms ....................................................... 16
    2.3.4 Solution sequence modifiers ................................. 20
    2.3.5 Semantics of SPARQL graph pattern expressions ........ 23
  2.4 Jena framework ....................................................... 26

3 Related Work 27
  3.1 Ontology mapping ................................................... 27
  3.2 SPARQL query rewriting ........................................... 28

4 Ontology mapping model 31
  4.1 Motivating example ................................................ 31
  4.2 Abstract syntax and semantics .................................. 37
  4.3 Ontology mapping types ........................................... 44
  4.4 Mapping representation ............................................ 45
    4.4.1 General structure ............................................ 47
    4.4.2 Mapping representation examples .................... 51

5 SPARQL query rewriting overview 55

6 Data Triple Pattern rewriting 59
  6.1 Rewriting based on triple pattern's subject part .......... 60
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2</td>
<td>Rewriting based on triple pattern’s object part</td>
<td>61</td>
</tr>
<tr>
<td>6.3</td>
<td>Rewriting based on triple pattern’s predicate part</td>
<td>68</td>
</tr>
<tr>
<td>6.4</td>
<td>Combination examples</td>
<td>73</td>
</tr>
<tr>
<td>7</td>
<td>Schema Triple Pattern rewriting</td>
<td>81</td>
</tr>
<tr>
<td>7.1</td>
<td>Rewriting based on triple pattern’s subject part</td>
<td>86</td>
</tr>
<tr>
<td>7.2</td>
<td>Rewriting based on triple pattern’s object part</td>
<td>93</td>
</tr>
<tr>
<td>8</td>
<td>Graph pattern rewriting</td>
<td>101</td>
</tr>
<tr>
<td>9</td>
<td>Implementation</td>
<td>113</td>
</tr>
<tr>
<td>10</td>
<td>Conclusion</td>
<td>115</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>116</td>
</tr>
<tr>
<td>A</td>
<td>Semantics of property relationships</td>
<td>121</td>
</tr>
<tr>
<td>A.1</td>
<td>Equivalence/subsumption between properties in OWL</td>
<td>121</td>
</tr>
<tr>
<td>A.2</td>
<td>Equivalence/subsumption between properties in our framework</td>
<td>122</td>
</tr>
<tr>
<td>B</td>
<td>Data Triple Pattern rewriting correctness</td>
<td>123</td>
</tr>
<tr>
<td>B.1</td>
<td>Proof of Lemma 6.1</td>
<td>125</td>
</tr>
<tr>
<td>B.2</td>
<td>Proof of Lemma 6.2</td>
<td>126</td>
</tr>
<tr>
<td>B.3</td>
<td>Proof of Lemma 6.3</td>
<td>132</td>
</tr>
<tr>
<td>C</td>
<td>Mapping representation ontology</td>
<td>143</td>
</tr>
<tr>
<td>D</td>
<td>Mapping representation example</td>
<td>155</td>
</tr>
</tbody>
</table>
List of Tables

Table 2.1 Notation used for the definition of SPARQL graph pattern semantics 24
Table 4.1 Class constructors ........................................... 38
Table 4.2 Class constructors (continued from Table 4.1) ............... 39
Table 4.3 Object property constructors .................................. 40
Table 4.4 Object property constructors (continued from Table 4.3) .... 41
Table 4.5 Datatype property constructors ............................... 41
Table 4.6 Datatype property constructors (continued from Table 4.5) .. 42
Table 4.7 Terminological and assertional axioms .......................... 43
Table 4.8 Class mapping examples ...................................... 46
Table 4.9 Object property mapping examples ............................ 46
Table 4.10 Datatype property mapping examples .......................... 46
Table 4.11 Individual mapping examples .................................. 46
Table 5.1 Triple pattern categorization example .......................... 56
Table 6.1 Notation used for the Data Triple Pattern rewriting functions .. 60
Table 7.1 Class axioms used for the rewriting of Schema Triple Patterns ...
Table 7.2 Property axioms used for the rewriting of Schema Triple Patterns ...
Table 7.3 Notation used for the Schema Triple Pattern rewriting functions ....
Table B.1 Notation used for Data triple pattern rewriting proofs ...........
List of Figures

Figure 2.1 RDF graph representation example .............................. 6
Figure 4.1 Semantically Overlapping Ontologies .......................... 32
Figure 4.2 Class mapping using an equivalence relationship ............ 33
Figure 4.3 Class mapping using a subsumption relationship ............. 33
Figure 4.4 Class mapping using a union operation ....................... 33
Figure 4.5 Class mapping using union and intersection operations ... 34
Figure 4.6 Class mapping using a property restriction ................. 34
Figure 4.7 Class mapping using property restrictions ................... 35
Figure 4.8 Mapping between two individuals ............................. 35
Figure 4.9 Mapping between two datatype properties ................... 35
Figure 4.10 Object property mapping using a domain restriction ........ 36
Figure 4.11 Object property mapping using an inverse operation ...... 36
Figure 4.12 Datatype property mapping using a union operation ...... 37
Figure 4.13 Datatype property mapping using a composition operation 37
Figure 9.1 System reference architecture ................................. 113
Chapter 1

Introduction

Data access from distributed autonomous web resources needs to take into account the data semantics at the conceptual level. Assuming that the resources are organized and accessed with the same model and language, a straightforward approach to semantic interoperability is to adhere to a common conceptualization (i.e. a global ontological conceptualization). However, in real-world environments, independent institutions often do not adhere to common standards. Attempts to find an agreement for a common conceptualization often results in semantically weak minimum consensus schemes (e.g. the Dublin Core [28]) or models with extensive and complex semantics (e.g. the CIDOC/CRM [12]). Moreover, it is not often feasible for cooperating institutions to agree on a certain model or apply an existing standard because they often already have their own proprietary conceptualizations. In this environment, query mediation over mapped ontologies has become a major research challenge since it allows uniform semantic information retrieval and at the same time permits diversification on individual conceptualizations followed by distributed federated information sources.

A mediator architecture is a common approach in information integration systems [42]. Mediated query systems represent a uniform data access solution by providing a single point for querying access to various data sources. A mediator contains a global query processor which is used to send sub-queries to local data sources. The local query results are then combined and returned back to the query processor. Its main benefit is that the query formulation process becomes independent of the mediated data sources requiring from end-users to be aware only of their own conceptualization of the knowledge domain.

In this thesis, we describe a mediator based methodology and system for integrating information from federated OWL/RDF knowledge bases. The mediator uses mappings between the OWL [3] ontology of the mediator (global ontology) and the federated site ontologies (local ontologies). SPARQL [33] queries posed over the mediator, are decomposed and rewritten in order to be submitted over the federated sites. The SPARQL
queries are locally evaluated and the results are returned to the mediator site. In this thesis we focus on the following research issues:

- determination of the different mapping types, which can be used in SPARQL query rewriting
- modeling of the mappings between the global ontology and the local ontologies
- rewriting of the SPARQL queries posed over the global ontology in terms of the local ontologies

Regarding the task of mapping modeling, the focus of this work is on the semantics and syntax of the ontology executable mappings. For identifying and describing such mappings, we define a formal grammar for mapping definition.

Based on these mapping types we provide a complete set of graph pattern rewriting functions that cover all the SPARQL grammar variations and can be used in the process of query rewriting for local ontologies. These functions are generic and can be used for SPARQL query rewriting over any overlapping ontology set. We show that the provided functions are semantics preserving, in the sense that each rewriting step that we perform (in order to rewrite the initial query) preserves the mapping semantics.

Contribution. The main contributions of this thesis are summarized as follows:

- A model for the expression of mappings between OWL DL ontologies in the context of SPARQL query rewriting. This mapping model consists of a formal grammar for the mapping definition and a formal specification of the mappings semantics.
- A generic formal methodology for the SPARQL query rewriting process, based on a set of mappings between OWL ontologies.
- A system implementation of the proposed methodology.

Outline. The rest of this thesis is organized as follows: An introduction to the standards and the technologies used for this thesis is presented in Chapter 2. The related work is discussed in Chapter 3. The mapping model which has been developed in order to express the mappings between the OWL ontologies is described in Chapter 4. The SPARQL query rewriting process is described comprehensively in Chapters 5, 6, 7 and 8. The implementation of the system that supports the query rewriting is discussed in Chapter 9. Finally, Chapter 10 concludes our work.
Chapter 2

Background

In this chapter we present the standards used in this thesis, as well as the technologies used for the implementation of our SPARQL query rewriting framework. Section 2.1 presents RDF/S, the standard language for representing information about resources in the World Wide Web. Section 2.2 presents OWL, the standard language for defining and instantiating Web ontologies. Section 2.3 presents SPARQL, the standard query language for RDF. Finally, Section 2.4 presents Jena, an open source Java framework for building Semantic Web applications.

From this point forward we consider the following namespaces:

- RDF namespace: rdf = http://www.w3.org/1999/02/22-rdf-syntax-ns#
- RDF Schema namespace: rdfs = http://www.w3.org/2000/01/rdf-schema#
- XML Schema namespace: xsd = http://www.w3.org/2001/XMLSchema#
- OWL namespace: owl = http://www.w3.org/2002/07/owl#
- Bookstore namespace (used in examples): ns = http://example.org/Bookstore#

2.1 Resource Description Framework and Schema Language (RDF/S)

The Resource Description Framework (RDF) [27] is the standard language for representing information about resources in the World Wide Web. RDF is based on the idea of identifying things using Web identifiers (called Internationalized Resource Identifiers, or IRIs [13]). IRIs are a generalization of URIs and are fully compatible with URIs and URLs.
The atomic constructs of RDF are statements, which are triples (subject, predicate, object) consisting of the resource (the subject) being described, a property (the predicate), and a property value (the object).

**Example 2.1.** The assertion of the following RDF triples mean that the resource book1, under the namespace ns, has a property title under the same namespace, with value “Database Systems”.

@prefix ns: <http://example.org/Bookstore#> .

It is worth to mention that in triple format IRI references are designated using the ‘<’ and ‘>’ delimiters. Moreover, the @prefix keyword associates a prefix label with an IRI. A prefixed name is a prefix label and a local part, separated by a colon ‘:’. A prefixed name is mapped to an IRI by concatenating the IRI associated with the prefix and the local part.

RDF can be expressed in a variety of formats including RDF/XML. However, in this thesis we use the triple form. Data values in RDF are represented by so-called literals. The value of every literal is generally described by a sequence of characters. The interpretation of such sequences is determined based on a given datatype (XML Schema datatype mainly). In triple form, the syntax for literals is a string (enclosed in double quotes, "..."), with an optional datatype IRI (introduced by ‘^-’).

**Example 2.2.** The assertion of the following RDF triples mean that the resource book1 under the namespace ns, has a property price under the same namespace, with value 27.

@prefix ns: <http://example.org/Bookstore#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

The xsd:integer part of the literal "27"^^xsd:integer is the XML Schema datatype for integers.

Resources in RDF may be anonymous (i.e. not identified by an IRI). Such resources are called blank nodes. In triple form, a blank node is indicated by the label form, such as “_:abc”.

**Definition 2.1 (RDF Triple).** Let $I$ be the set of IRIs, $L$ be the set of the RDF Literals, and $B$ be the set of the blank nodes. A triple $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$ is called an RDF triple, where $s$, $p$, and $o$ are a subject, predicate, and object, respectively.
Example 2.3. The assertion of the following RDF triples mean that something has an author whose value is “Jeffrey D. Ullman”.

@example
@prefix ns: <http://example.org/Bookstore#> .
_:a ns:author "Jeffrey D. Ullman" .
@end

A collection of RDF statements (RDF triples) can be intuitively understood as a directed labeled graph, where resources are nodes and statements are arcs (from the subject node to the object node) connecting the nodes. It is worth to mention that a relational data model is easily mapped into this form, with a node corresponding to a table row or primitive value, and an arc corresponding to a column identifier.

Definition 2.2 (RDF Graph). An RDF graph \( G \) is a set of RDF triples.

Example 2.4. The assertion of the following RDF triples means that “Jeffrey D. Ullman” is an author of a book entitled “Database Systems” whose publisher is “Prentice Hall”.

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

@end

Figure 2.1 shows the representation of the above RDF triples as a directed graph. Moreover, the graph of Figure 2.1 can be represented as a tuple in a relational data model, as follows:

<table>
<thead>
<tr>
<th>ns:Publication</th>
<th>id</th>
<th>ns:title</th>
<th>ns:author</th>
<th>ns:publisher</th>
</tr>
</thead>
<tbody>
<tr>
<td>ns:book1</td>
<td>&quot;Database Systems&quot;</td>
<td>&quot;Jeffrey D. Ullman&quot;</td>
<td>&quot;Prentice Hall&quot;</td>
<td></td>
</tr>
</tbody>
</table>

RDF provides a number of additional capabilities, such as built-in types and properties for representing groups of resources and simple RDF statements. These types and properties are described using a set of reserved words (prefixed \( \text{http://www.w3.org/1999/02/22-rdf-syntax-ns#} \)) called the RDF vocabulary. However, RDF user communities also need the ability to define the vocabularies (terms) they intend to use in those statements,
Figure 2.1: RDF graph representation example.

specifically, to indicate that they are describing specific kinds or classes of resources, and to use specific properties in describing those resources. Consequently, the RDFS vocabulary came to build on the limited vocabulary of RDF.

RDFS (RDF Schema) [27], [8] is an extension of RDF designed to describe relationships between resources and/or resources using a set of reserved words (prefixed http://www.w3.org/2000/01/rdf-schema#) called the RDFS vocabulary. It describes constructs for types of objects (classes), relating types to one another (subclasses), properties that describe objects (properties), and relationships between them (subproperty). The class system in RDFS includes a simple notion of inheritance, based on set inclusion. For example, one class is a subclass of another means that instances of the one are also instances of the other.

A class in RDFS corresponds to the generic concept of a type or category, somewhat like the notion of a class in object-oriented programming languages such as Java, and is defined using the construct rdfs:Class. RDF classes can be used to represent almost any category of resources, such as Web pages, people, document types, databases or abstract concepts. The resources that belong to a class are called its instances. Classes can be organized in a hierarchical fashion using the construct rdfs:subClassOf.

A property in RDFS is used to characterize a class/classes and is defined using the construct rdf:Property. The RDFS also provides a vocabulary used to describe how properties and classes are intended to be used together in RDF data. This kind of information is supplied by using the rdfs:domain and rdfs:range constructs. Similarly to classes, RDFS provides a way to specialize properties by using the construct rdfs:subPropertyOf.
Moreover, RDFS provides a number of other built-in properties which can be used to provide documentation and other information about an RDF Schema or about instances. For example, the construct \texttt{rdfs:comment} can be used to provide a human-readable description of a resource, while the construct \texttt{rdfs:label} can be used to provide a more human-readable version of a resource’s name.

Finally, the semantics of RDFS is expressed through the mechanism of inferencing (i.e. the meaning of any construct in RDFS is given by the inferences that can be inferred from it).

2.2 Web Ontology Language (OWL)

OWL [3] is the standard language for defining and instantiating Web ontologies. OWL and RDFS are much of the same thing, but OWL is a stronger language with greater machine interpretability than RDFS. Moreover, OWL comes with a larger vocabulary and stronger syntax than RDFS. For example, OWL provides constructs to define property restrictions using value/cardinality constraints, as well as constructs to define complex classes using basic set operations (union, intersection and complement). It provides three increasingly expressive sublanguages designed for use by specific communities of implementers and users. These sublanguages are characterised by formal semantics and RDF/XML-based serializations for the Semantic Web.

- **OWL Lite** supports those users primarily needing a classification hierarchy and simple constraints. OWL Lite has a lower formal complexity than OWL DL. For example, it does not support the definition of complex classes using the union/complement operations. Moreover, while it supports cardinality constraints, it only permits cardinality values of 0 or 1.

- **OWL DL** supports those users who want the maximum expressiveness while retaining computational completeness (all conclusions are guaranteed to be computable) and decidability (all computations will finish in finite time). OWL DL includes all OWL language constructs, but they can be used only under certain restrictions (for example, while a class may be a subclass of many classes, a class cannot be an instance of another class). OWL DL is so named due to its correspondence with Description Logics, a field of research that has studied the logics that form the formal foundation of OWL.

- **OWL Full** is meant for users who want maximum expressiveness and the syntactic freedom of RDF with no computational guarantees. For example, in OWL Full a class can be treated simultaneously as a collection of individuals and as an individual in its own right. It is unlikely that any reasoning software will be able to support every feature of OWL Full.
Each of these sublanguages is an extension of its simpler predecessor, both in what can be legally expressed and in what can be validly concluded. The following set of relations hold, while their inverses do not.

- Every valid OWL Lite ontology is a valid OWL DL ontology.
- Every valid OWL DL ontology is a valid OWL Full ontology.
- Every valid OWL Lite conclusion is a valid OWL DL conclusion.
- Every valid OWL DL conclusion is a valid OWL Full conclusion.

Languages in the OWL family are capable of creating classes, properties, defining instances and its operations. The resources in the OWL vocabulary have IRI references with the prefix http://www.w3.org/2002/07/owl#.

An individual is an object. It corresponds to a Description Logic individual. OWL provides mechanisms in order to declare two individuals to be identical or different, by using the owl:sameAs and owl:differentFrom constructors, respectively.

A class (defined by using the construct owl:Class) is a collection of objects. It corresponds to a Description Logic (DL) concept and may contain any number of individuals, instances of the class. An individual may belong to none, one or more classes. A class may be defined to be subclass of another (using the construct rdfs:subClassOf), inheriting characteristics from its parent superclass. This corresponds to logical subsumption and DL concept inclusion. All classes are subclasses of owl:Thing (DL top notated $\top$), the root class. All classes are subclassed by owl:Nothing (DL bottom notated $\bot$), the empty class. Similarly, two classes may be defined to be equivalent (using the construct owl:equivalentClass), indicating that these two classes have precisely the same instances. This corresponds to logical equivalence and DL concept equality.

**Example 2.5.** Let Product be an OWL class and let HelloweenAudioCD be an instance of Product. Similarly, let Book be an OWL class and let DatabaseSystems be an instance of Book. The RDF/XML syntax used to define these statements is provided below.

```xml
<owl:Class rdf:ID="Product"/>
<owl:Class rdf:ID="Book"/>

<Product rdf:ID="HelloweenAudioCD"/>
<Book rdf:ID="DatabaseSystems"/>
```

**Example 2.6.** Consider the classes and the instances defined in the previous example. Moreover, let the class Book be subclass of the class Product. The RDF/XML syntax used to define this statement is provided below.
By defining that the class Book is a subclass of the class Product, we implicitly infer that every instance of the class Book is also an instance of the class Product. Consequently, DatabaseSystems which has been defined as an instance of the class Book, is also an instance of the class Product.

OWL provides additional constructors with which to form classes. These constructors can be used to create so-called class expressions. OWL supports the basic set operations, namely union, intersection and complement. These are named owl:unionOf, owl:intersectionOf, and owl:complementOf, respectively. Additionally, classes can be enumerated by defining explicitly the individuals which are members of a class (i.e. class extension). Class extensions can be stated explicitly by means of the owl:oneOf constructor. Furthermore, it is possible to assert that class extensions must be disjoint (using the construct owl:disjointWith).

A property is a directed binary relation that specifies class characteristics. It corresponds to a Description Logic role. A property may be defined to be subproperty of another (using the construct rdfs:subPropertyOf), inheriting characteristics from its parent superproperty. Similarly, two properties may be defined to be equivalent, using the construct owl:equivalentProperty. Two types of properties are distinguished:

- **Datatype properties** are relations between instances of classes (i.e. individuals) and RDF literals or XML schema datatypes. A datatype property is defined by using the construct owl:DatatypeProperty.

- **Object properties** are relations between instances of two classes (i.e. individuals). An object property is defined by using the construct owl:ObjectProperty.

Properties may possess logical capabilities such as being transitive, symmetric, inverse and functional. Properties may also have domains and ranges. It is possible to constrain the range of a property in specific contexts in a variety of ways, by using either cardinality or value restrictions.

**Example 2.7.** Consider the classes and the instances defined in the previous example. Let the class Product have a datatype property price. This infers that the instances of
the classes Product and Book can be described using the datatype property price, since Book is a subclass of Product. Furthermore, let the datatype property price range over the XML Schema datatype xsd:decimal. The RDF/XML syntax used to define these statements is provided below.

<owl:Class rdf:ID="Product"/>
<owl:Class rdf:ID="Book">
  <rdfs:subClassOf rdf:resource="#Product"/>
</owl:Class>

<owl:DatatypeProperty rdf:ID="price">
  <rdfs:domain rdf:resource="#Product"/>
  <rdfs:range rdf:resource="&xsd;decimal"/>
</owl:DatatypeProperty>

<Product rdf:ID="HelloweenAudioCD">
  <price rdf:datatype="&xsd;decimal">15.30</price>
</Product>
<Book rdf:ID="DatabaseSystems">
  <price rdf:datatype="&xsd;decimal">57.90</price>
</Book>

**Example 2.8.** Consider the classes, the properties and the instances defined in the previous example. Let the class Book have an object property publisher. Moreover, let the object property publisher range over an OWL class Publisher. Similarly, let an object property publishes be the inverse of the object property publisher, having as domain the instances of the class Publisher and as range the instances of the class Book. The RDF/XML syntax used to define these statements is provided below.

<owl:Class rdf:ID="Product"/>
<owl:Class rdf:ID="Book">
  <rdfs:subClassOf rdf:resource="#Product"/>
</owl:Class>
<owl:Class rdf:ID="Publisher"/>

<owl:DatatypeProperty rdf:ID="price">
  <rdfs:domain rdf:resource="#Product"/>
  <rdfs:range rdf:resource="&xsd;decimal"/>
</owl:DatatypeProperty>
2.3. SPARQL query language

This section presents SPARQL [33], the standard language for querying RDF data. Section 2.3.1 presents the syntax used by SPARQL for RDF terms and triple patterns, while Section 2.3.2, Section 2.3.3 and Section 2.3.4 present the SPARQL graph patterns, the query forms and the solution sequence modifiers of SPARQL, respectively. Finally, Section 2.3.5 presents the semantics of SPARQL graph pattern expressions based on [32].

2.3.1 SPARQL syntax

This section presents the syntax used by SPARQL for RDF terms and triple patterns.

Syntax for IRIs

IRI references are designated using the '<' and '>' delimiters. The PREFIX keyword can be used to associate a prefix label with an IRI. A prefixed name is a prefix label and a local part, separated by a colon ':'. A prefixed name is mapped to an IRI by concatenating the IRI associated with the prefix and the local part. The prefix label or the local part may be empty.
The following fragments are some of the different ways to write the same IRI:

1. `<http://example.org/Bookstore#DatabaseSystems>`

2. `PREFIX ns: <http://example.org/Bookstore#>
   ns:DatabaseSystems`

3. `PREFIX : <http://example.org/Bookstore#>
   :DatabaseSystems`

Syntax for literals

The general syntax for literals is a string (enclosed in either double quotes, "....", or single quotes, '....'), with either an optional language tag (introduced by @) or an optional datatype IRI or prefixed name (introduced by ^). As a convenience, integers can be written directly (without quotation marks and an explicit datatype IRI) and are interpreted as typed literals of the XML Schema datatype `xsd:integer`. Furthermore, decimal numbers for which there is '.' in the number but no exponent are interpreted as `xsd:decimal` and numbers with exponents are interpreted as `xsd:double`. Values of type `xsd:boolean` can also be written as `true` or `false`.

Examples of literal syntax in SPARQL include:

1. "chat"

2. 'chat'@fr with language tag "fr"

3. "xyz"^^<http://example.org/ns/userDatatype>

4. 1, which is the same as "1"^^xsd:integer

5. 1.3, which is the same as "1.3"^^xsd:decimal

6. 1.0e6, which is the same as "1.0e6"^^xsd:double

7. true, which is the same as "true"^^xsd:boolean

8. false, which is the same as "false"^^xsd:boolean

Syntax for query variables

Query variables in SPARQL queries have global scope. Consequently, the use of a given variable name anywhere in a query identifies the same variable. Variables are prefixed by either "?" or "$". These two symbols are not considered part of the variable’s name. In a query, $abc and ?abc identify the same variable.
2.3. SPARQL QUERY LANGUAGE

Syntax for blank nodes

Blank nodes in SPARQL queries act as non-distinguished variables and not as references to specific blank nodes in the data being queried. Blank nodes are indicated by the label form, such as "_:abc".

Syntax for triple patterns

Let $I$ be the set of IRIs, $L$ be the set of the RDF Literals, and $B$ be the set of the blank nodes. Assume additionally the existence of an infinite set $V$ of variables disjoint from the previous sets ($I$, $B$, $L$).

**Definition 2.3 (Triple pattern).** A triple $(s; p; o) \in (I \cup L \cup V) \times (I \cup V) \times (I \cup L \cup V)$ is called a triple pattern, where $s$, $p$, and $o$ are a subject, predicate, and object, respectively.

Triple Patterns are written as a whitespace-separated list of a subject, predicate and object. Triple patterns are like RDF triples except that each of the subject, predicate and object may be a variable. Some triple pattern examples are the following:

   The above triple pattern contains the resource identifier `http://example.org/Bookstore#DatabaseSystems` in its subject part, the property `title` under the prefix `ns` in its predicate part and the variable `title` in its object part.

2. `?x foaf:name "Kostas"
   The above triple pattern contains the variable `x` in its subject part, the property `name` under the prefix `foaf` in its predicate part and the literal `Kostas` in its object part.

3. `?x ?y "55.30"^^xsd:decimal`
   The above triple pattern contains the variable `x` in its subject part, the variable `y` in its predicate part and the literal `"55.30"^^xsd:decimal` in its object part.

4. `?s ?p ?o`
   The above triple pattern contains the variable `s` in its subject part, the variable `p` in its predicate part and the variable `o` in its object part.

2.3.2 Graph patterns

SPARQL is based around graph pattern matching. More complex graph patterns can be formed by combining smaller patterns in various ways.
Definition 2.4 (Graph pattern). A SPARQL graph pattern expression is defined recursively as follows:

- A triple pattern is a graph pattern.
- If $P_1$ and $P_2$ are graph patterns, then expressions $(P_1 \text{ AND } P_2)$, $(P_1 \text{ OPT } P_2)$, and $(P_1 \text{ UNION } P_2)$ are graph patterns (group graph pattern, optional graph pattern, and alternative graph pattern, respectively).
- If $P$ is a graph pattern and $R$ is a SPARQL built-in condition, then the expression $(P \text{ FILTER } R)$ is a graph pattern (a filter graph pattern).

We note that a SPARQL built-in condition is constructed using IRIs, RDF literals, variables and constants, as well as logical connectives (and - &&, or - ||, not - !), operators ($=$, $!=$, $>$, $<$, $\geq$, $\leq$, $+$, $-$, $\ast$, $/$) and built-in functions (e.g. bound, isIRI, isLiteral, datatype, lang, str, regex).

In the rest of this subsection we analyze the different types of SPARQL graph pattern expressions.

Basic graph patterns

Basic graph patterns are sets of triple patterns. SPARQL graph pattern matching is defined in terms of combining the results from matching basic graph patterns. A sequence of triple patterns interrupted by a filter comprises a single basic graph pattern. Filters are constraints expressed by the keyword FILTER, which are used to restrict the graph pattern solutions to those for which the filter expression evaluates to true.

A filter is consisted of a SPARQL built-in condition, which is constructed using IRIs, RDF literals, variables and constants, as well as logical connectives (and - &&, or - ||, not - !), operators ($=$, $!=$, $>$, $<$, $\geq$, $\leq$, $+$, $-$, $\ast$, $/$) and built-in functions (e.g. bound, isIRI, isLiteral, datatype, lang, str, regex).

Definition 2.5 (Basic graph pattern). A finite sequence of conjunctive triple patterns (separated with “.”) and possible filters is called basic graph pattern.

Some basic graph pattern examples are the following:

1. $\text{?x ns:title ?title .}$
   $\text{?x ns:price "62"^^xsd:decimal .}$
2. $\text{?x ns:title "Database Systems" .}$
   $\text{?x ns:price ?price .}$
   $\text{FILTER(?price>50)}$
Group graph patterns

The group graph pattern is the most general form of a graph pattern, since it may contain every other graph pattern type. A group graph pattern is delimited with braces ({}). A group graph pattern example is the following:

```
{ ?x ns:title "Database Systems" .
  ?x ns:author ?author . }
```

The above group graph pattern is consisted of two triple patterns and is considered to be equivalent with the following:

```
{ { ?x ns:title "Database Systems" .}
  { ?x ns:author ?author . } }
```

A constraint, expressed by the keyword FILTER, is a restriction on the solutions over the whole group in which the filter appears. The following patterns all have the same solutions:

   ?x ns:price ?price .
   FILTER (?price<60) }
```

2. `{ FILTER (?price<60)
   ?x ns:price ?price . }
```

   FILTER (?price<60)
   ?x ns:price ?price . }
```

Optional graph patterns

Basic graph patterns allow applications to make queries where the entire query pattern must match for there to be a solution. However, it is useful to be able to have queries that allow information to be added to the solution where the information is available, but do not reject the solution because some part of the query pattern does not match. Optional matching provides this facility: if the optional part does not match, it creates no bindings but does not eliminate the solution.

Optional parts of a graph pattern may be specified syntactically with the OPTIONAL keyword applied to a graph pattern. A graph pattern example that contains an optional part is the following:

```
?x foaf:name ?name .
OPTIONAL{ ?x foaf:mbox ?mbox }
```
In an optional match, either the optional graph pattern matches a graph, thereby defining and adding bindings to one or more solutions, or it leaves a solution unchanged without adding any additional bindings. The above graph pattern matches the names of people in the data. If there is a triple with predicate \texttt{foaf:mbox} and the same subject, a solution will contain the object of that triple as well. The entire optional graph pattern must match for the optional graph pattern to affect the query solution.

**Alternative graph patterns**

SPARQL provides a means of combining graph patterns so that one of several alternative graph patterns may match. If more than one of the alternatives matches, all the possible pattern solutions are found. Pattern alternatives are syntactically specified with the \texttt{UNION} keyword. An alternative graph pattern example is the following:

\[
\]

The above graph pattern matches titles and books in the data, whether the property title is under the namespace \texttt{ns1} or \texttt{ns2}.

### 2.3.3 Query forms

SPARQL has four query forms. These query forms use the solutions from pattern matching to form result sets or RDF graphs. The query forms are:

**Select**

The \texttt{SELECT} query form returns variables and their bindings directly. The syntax \texttt{SELECT *} is an abbreviation that selects all of the variables in a query.

**Example 2.9.** Consider the query posed over an RDF dataset: “Return the titles and the prices of books written by Jeffrey D. Ullman”.

- RDF data:

  ```
  @prefix ns: <http://example.org/Bookstore#> .
  @prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

  _:a ns:title "Database Systems" .
  _:a ns:author "Jeffrey D. Ullman" .
  _:a ns:price "62"^^xsd:decimal .
  _:b ns:title "Foundations of Computer Science" .
  _:b ns:author "Jeffrey D. Ullman" .
  _:b ns:price "35.40"^^xsd:decimal .
  ```
Construct

The CONSTRUCT query form returns an RDF graph constructed by substituting variables in a set of triple templates. If any instantiation produces a triple containing an unbound variable or an illegal RDF construct, such as a literal in subject or predicate position, then that triple is not included in the output RDF graph.

Example 2.10. Consider the query posed over an RDF dataset: “Return an RDF graph based on the foaf vocabulary, while the queried RDF dataset is based on the ns vocabulary”.

- RDF data:

```sparql
@prefix ns: <http://example.org/Bookstore#> .
_:a ns:title "Database Systems" .
_:a ns:author _:b .
_:b ns:givenName "Jeffrey" .
_:b ns:familyName "D. Ullman" .
_:c ns:title "Ontology Matching" .
```
Query:

```sparql
@PREFIX ns: <http://example.org/Bookstore#> .
@PREFIX foaf: <http://xmlns.com/foaf/0.1/> .

CONSTRUCT {?x foaf:firstname ?gname .  
            ?x foaf:surname  ?fname .}
WHERE {?x ns:givenName  ?gname .  
       ?x ns:familyName  ?fname .}
```

Results:

```sparql
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
_:e foaf:firstname "Jeffrey" .  
_:e foaf:surname  "D. Ullman" .  
_:f foaf:firstname "Jérôme" .  
_:f foaf:surname  "Euzenat" .
```

Ask

The ASK query form returns no information about the possible query solutions, just whether or not a solution exists.

**Example 2.11.** Consider the query posed over an RDF dataset: “Return whether there are any cheap books (i.e. price lower than 30) or not”.

- RDF data:

```sparql
@prefix ns: <http://example.org/Bookstore#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

_:a ns:title  "Database Systems" .  
_:a ns:author "Jeffrey D. Ullman" .  
_:a ns:price "62"^^xsd:decimal .  
_:b ns:title  "Ontology Matching" .  
_:b ns:author "Jérôme Euzenat" .  
_:b ns:price "55.30"^^xsd:decimal .
```
2.3. SPARQL QUERY LANGUAGE

• Query:

@PREFIX ns: <http://example.org/Bookstore#> .
ASK {?x ns:price ?price .
    FILTER (?price < 30)}

• Results: no

Describe

The DESCRIBE query form returns a single result RDF graph containing RDF data about resources. This data is not prescribed by a SPARQL query, where the query client would need to know the structure of the RDF in the data source, but, instead, is determined by the SPARQL query processor. The DESCRIBE query form takes each of the resources identified in a solution, together with any resources directly named by using an IRI, and assembles a single RDF graph by taking a “description” which can come from any information available including the target RDF dataset. The syntax DESCRIBE * is an abbreviation that describes all of the variables in a query.

Example 2.12. Consider the query posed over an RDF dataset: “Describe a resource with title Database Systems”.

• RDF data:

@prefix ns: <http://example.org/Bookstore#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
_:a ns:title "Database Systems" .
_:a ns:author "Jeffrey D. Ullman" .
_:a ns:price "62"^^xsd:decimal .
_:b ns:title "Ontology Matching" .
_:b ns:author "Jérôme Euzenat" .
_:b ns:price "55.30"^^xsd:decimal .

• Query:

@PREFIX ns: <http://example.org/Bookstore#> .

DESCRIBE ?x
WHERE {?x ns:title ?title .
    FILTER (?title = "Database Systems")}
2.3.4 Solution sequence modifiers

Query patterns generate an unordered collection of solutions. Each solution is a partial function from variables to RDF terms. These solutions are initially treated as a sequence (a solution sequence), in no specific order. In case that any sequence modifiers are applied, a new sequence is created. Finally, this latter sequence is used to generate one of the results of a SPARQL query form.

The solution sequence modifiers may be applied on the SELECT, CONSTRUCT and DESCRIBE query forms. In addition, the DISTINCT and REDUCE modifiers may be applied only on the SELECT query form.

Order by

The ORDER BY clause establishes the order of a solution sequence. Furthermore, it is possible to use the ASC() and DESC() modifiers in order to specify whether the order should be ascending or descending. In case that any ASC() or DESC() modifier has been specified, the order of the solution sequence is ascending.

For example, the following query lists the solution sequence firstly in ascending order, based on the values of the variable title. Regarding the solutions having the same title value, they are listed in descending order based on the values of the variable price.

```
@prefix ns: <http://example.org/Bookstore#> .
SELECT ?title ?price
WHERE {?x ns:title ?title .
   ?x ns:price ?price .}
ORDER BY ?title DESC (?price)
```

Distinct

The DISTINCT clause eliminates duplicate solutions. For example, consider the following queries posed over the same RDF dataset.
• RDF data:

@prefix ns: <http://example.org/Bookstore#> .

_:a ns:title "Database Systems" .
_:a ns:author "Jeffrey D. Ullman" .
_:b ns:title "Foundations of Computer Science" .
_:b ns:author "Jeffrey D. Ullman" .
_:c ns:title "Ontology Matching" .
_:c ns:author "Jérôme Euzenat" .

• Query 1:

@PREFIX ns: <http://example.org/Bookstore#> .

SELECT ?author
WHERE {?x ns:author ?author}

• Query 1 Results:

<table>
<thead>
<tr>
<th>?author</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Jeffrey D. Ullman&quot;</td>
</tr>
<tr>
<td>&quot;Jeffrey D. Ullman&quot;</td>
</tr>
<tr>
<td>&quot;Jérôme Euzenat&quot;</td>
</tr>
</tbody>
</table>

• Query 2:

@PREFIX ns: <http://example.org/Bookstore#> .

SELECT DISTINCT ?author
WHERE {?x ns:author ?author}

• Query 2 Results:

<table>
<thead>
<tr>
<th>?author</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Jeffrey D. Ullman&quot;</td>
</tr>
<tr>
<td>&quot;Jérôme Euzenat&quot;</td>
</tr>
</tbody>
</table>
Reduced

The REDUCED clause permits a specific number of duplicate solutions. This number is specified by the SPARQL query engine that executes the query and is at least one and not more than the cardinality of the solution set with no DISTINCT or REDUCED modifier. For example, consider the following query posed over the data provided below.

- RDF data:

```sparql
@prefix ns: <http://example.org/Bookstore#> .
_:a ns:title "Database Systems" .
_:a ns:author "Jeffrey D. Ullman" .
_:b ns:title "Foundations of Computer Science" .
_:b ns:author "Jeffrey D. Ullman" .
_:c ns:title "Introduction to Automata and Language Theory" .
_:c ns:author "Jeffrey D. Ullman" .
```

- Query:

```sparql
@PREFIX ns: <http://example.org/Bookstore#> .

SELECT REDUCED ?author
WHERE {?x ns:author ?author}
```

- Results: May have one, two (shown here) or three solutions.

```
?author
"Jeffrey D. Ullman"
"Jeffrey D. Ullman"
```

Limit

The LIMIT clause puts an upper bound on the number of solutions returned. If the number of actual solutions is greater than the limit, then at most the limit number of solutions will be returned. A LIMIT of zero has no effect.

For example, the following query returns the cheapest books (at most 5 solutions) written by "Jeffrey D. Ullman".

```sparql
@PREFIX ns: <http://example.org/Bookstore#> .

SELECT ?title ?price
```
WHERE {?x ns:title ?title .
   ?x ns:price ?price .
   FILTER (?y = "Jeffrey D. Ullman")}
ORDER BY ?price
LIMIT 5

Offset

The OFFSET clause causes the solutions generated to start after the specified number of solutions. An OFFSET of zero has no effect.

For example, the following query firstly lists the solutions in ascending order based on the variable price. Then it skips the initial 10 solutions and returns at most the following 5.

@PREFIX ns: <http://example.org/Bookstore#> .

SELECT ?title ?price
WHERE {?x ns:title ?title .
   ?x ns:price ?price .
   FILTER (?y = "Jeffrey D. Ullman")}
ORDER BY ?price
OFFSET 10
LIMIT 5

2.3.5 Semantics of SPARQL graph pattern expressions

In this section we provide an overview of the semantics of SPARQL graph pattern expressions defined in [32], considering a function-based representation of a graph pattern evaluation over an RDF dataset.

In order not to confuse, the notation and the terminology followed in this section is differentiated in some cases, compared to the notation and terminology followed in [32]. In Table 2.1 we provide the notation which is used for defining the semantics of SPARQL graph pattern expressions.

Definition 2.6 (SPARQL graph pattern solution). A graph pattern solution $\omega : V \rightarrow (I \cup B \cup L)$ is a partial function that assigns RDF terms of an RDF dataset to variables of a SPARQL graph pattern. The domain of $\omega$, $\text{dom}(\omega)$, is the subset of $V$ where $\omega$ is defined. The empty graph pattern solution $\omega_0$ is the graph pattern solution with empty domain. The SPARQL graph pattern evaluation result is a set $\Omega$ of graph pattern solutions $\omega$. 
Table 2.1: The notation which is used for defining the semantics of SPARQL graph pattern expressions.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V$</td>
<td>The set of variables.</td>
</tr>
<tr>
<td>$I$</td>
<td>The set of IRIs.</td>
</tr>
<tr>
<td>$B$</td>
<td>The set of blank nodes.</td>
</tr>
<tr>
<td>$L$</td>
<td>The set of RDF Literals.</td>
</tr>
<tr>
<td>$\omega$</td>
<td>A graph pattern solution $\omega : V \rightarrow (I \cup B \cup L)$.</td>
</tr>
<tr>
<td>$\text{dom}(\omega)$</td>
<td>Domain of a graph pattern solution $\omega$ (subset of $V$).</td>
</tr>
<tr>
<td>$\text{var}(t)$</td>
<td>The variables of a triple pattern $t$.</td>
</tr>
<tr>
<td>$\omega(t)$</td>
<td>The triple obtained by replacing the variables in triple pattern $t$ according to a graph pattern solution $\omega$ (abusing notation).</td>
</tr>
<tr>
<td>$\omega \models R$</td>
<td>A graph pattern solution $\omega$ satisfies a built-in condition $R$.</td>
</tr>
<tr>
<td>$[[\cdot]]$</td>
<td>Graph pattern evaluation function.</td>
</tr>
<tr>
<td>$\cdot\cdot\cdot$</td>
<td>Graph pattern solution-based join.</td>
</tr>
<tr>
<td>$\cdot\cdot\cdot$</td>
<td>Graph pattern solution-based left outer join.</td>
</tr>
<tr>
<td>$\cdot\cdot\cdot$</td>
<td>Graph pattern solution-based difference.</td>
</tr>
<tr>
<td>$\cdot\cdot\cdot$</td>
<td>Graph pattern solution-based projection.</td>
</tr>
<tr>
<td>$\cdot\cdot\cdot$</td>
<td>Graph pattern solution-based union.</td>
</tr>
<tr>
<td>$\cap$</td>
<td>Set intersection.</td>
</tr>
<tr>
<td>$\cdot\cdot\cdot$</td>
<td>SPARQL variables.</td>
</tr>
<tr>
<td>$\cdot\cdot\cdot$</td>
<td>SPARQL unary predicate.</td>
</tr>
<tr>
<td>AND, OPT, UNION, FILTER</td>
<td>SPARQL graph pattern operators.</td>
</tr>
<tr>
<td>$\neg, \vee, \wedge$</td>
<td>Logical not, or, and.</td>
</tr>
<tr>
<td>$\cdot\cdot\cdot$</td>
<td>Inequality/equality operators.</td>
</tr>
</tbody>
</table>
Two graph pattern solutions $\omega_1$ and $\omega_2$ are compatible when for all $x \in \text{dom}(\omega_1) \cap \text{dom}(\omega_2)$, it is the case that $\omega_1(x) = \omega_2(x)$. Furthermore, two graph pattern solutions with disjoint domains are always compatible, and the empty graph pattern solution $\omega_0$ is compatible with any other graph pattern solution.

Let $\Omega_1$ and $\Omega_2$ be sets of graph pattern solutions and $\mathcal{J}$ be a set of SPARQL variables. The join, union, difference, projection and left outer join operations between $\Omega_1$ and $\Omega_2$ are defined as follows:

- $\Omega_1 \uplus \Omega_2 = \{ \omega_1 \cup \omega_2 \mid \omega_1 \in \Omega_1, \omega_2 \in \Omega_2 \text{ are compatible graph pattern solutions} \}$,
- $\Omega_1 \cup \Omega_2 = \{ \omega \mid \omega \in \Omega_1 \text{ or } \omega \in \Omega_2 \}$,
- $\Omega_1 \setminus \Omega_2 = \{ \omega \in \Omega_1 \mid \text{ for all } \omega' \in \Omega_2, \omega \text{ and } \omega' \text{ are not compatible} \}$,
- $\pi_{\mathcal{J}}(\Omega_1) = \{ \omega \mid \omega' \in \Omega_1, \text{dom}(\omega) = \text{dom}(\omega') \cap \mathcal{J} \text{ and } \forall x \in \text{dom}(\omega), \omega(x) = \omega'(x) \}$,
- $\Omega_1 \upharpoonright \Omega_2 = (\Omega_1 \uplus \Omega_2) \cup (\Omega_1 \setminus \Omega_2)$

The semantics of SPARQL graph pattern expressions is defined as a function $[[\cdot]]_D$ which takes a graph pattern expression and an RDF dataset $D$ and returns a set of graph pattern solutions (see Definition 2.8). Refer to Definition 2.7 for the semantics of FILTER expressions, which can be part of a SPARQL graph pattern.

**Definition 2.7 (SPARQL FILTER expression evaluation).** Given a graph pattern solution $\omega$ and a built-in condition $R$, we say that $\omega$ satisfies $R$, denoted by $\omega \models R$, if:

1. $R$ is $\text{bound}(?x)$ and $?x \in \text{dom}(\omega)$;
2. $R$ is $?x \text{ cp } c,?x \in \text{dom}(\omega)$ and $\omega(?x) \text{ oprt } c$, where $\text{cp} \rightarrow = | \leq | \geq | < | >$;
3. $R$ is $?x \text{ cp } ?y,?x \in \text{dom}(\omega),?y \in \text{dom}(\omega)$ and $\omega(?x) \text{ cp } \omega(?y)$, where $\text{cp} \rightarrow = | \leq | \geq | < | >$;
4. $R$ is $(\neg R_1), R_1$ is a built-in condition, and it is not the case that $\omega \models R_1$;
5. $R$ is $(R_1 \lor R_2), R_1$ and $R_2$ are built-in conditions, and $\omega \models R_1$ or $\omega \models R_2$;
6. $R$ is $(R_1 \land R_2), R_1$ and $R_2$ are built-in conditions, $\omega \models R_1$ and $\omega \models R_2$. 

Definition 2.8 (SPARQL graph pattern evaluation). Let $D$ be an RDF dataset over $(I \cup B \cup L)$, $t$ a triple pattern, $P$, $P_1$, $P_2$ graph patterns and $R$ a built-in condition. The evaluation of a graph pattern over $D$, denoted by $[[\cdot]]_D$, is defined recursively as follows:

1. $[[t]]_D = \{\omega \mid \text{dom}(\omega) = \text{var}(t) \text{ and } \omega(t) \in D\}$
2. $[[P_1 \text{ AND } P_2]]_D = [[P_1]]_D \cup [[P_2]]_D$
3. $[[P_1 \text{ OPT } P_2]]_D = [[P_1]]_D \cup [[P_2]]_D$
4. $[[P_1 \text{ UNION } P_2]]_D = [[P_1]]_D \cup [[P_2]]_D$
5. $[[P \text{ FILTER } R]]_D = \{\omega \in [[P]]_D \mid \omega \models R\}$

For a detailed description of SPARQL semantics and for a complete set of illustrative examples, refer to [32].

2.4 Jena framework

Jena is an open source Java framework for building Semantic Web applications. It provides a programmatic environment for RDF, RDFS, OWL and SPARQL, as well as a rule-based inference engine. More specifically, the Jena framework includes: a RDF API, reading and writing RDF in various formats (RDF/XML, N3 and N-Triples), an OWL API, in-memory and persistent storage and a SPARQL query engine.
Chapter 3

Related Work

In the Semantic Web literature a number of ontology based mediator architectures have been proposed, for example [41], [29]. In the following sections we present the most relevant research to the issues discussed in this thesis.

3.1 Ontology mapping

Ontology mapping is the task of relating the vocabulary of two ontologies by defining a set of correspondences. The correspondences between different entities of the two ontologies are typically expressed using some axioms described in a specific mapping language. The discovery and specification of mappings between two ontologies, is a process which can be achieved in three ways:

- **Manually**: defined by an expert who has a very good understanding of the ontologies to be mapped.
- **Automatically**: using various matching algorithms and techniques which compute similarity measures between different ontology terms.
- **Semi-automatically**: using various matching algorithms and techniques, as well as user feedback.

Many strategies and tools that produce automatically or semi-automatically mappings have been proposed and have their performance analyzed ([20], [24], [38], [10], [17], [14], [31]). Although the automatic or semi-automatic techniques and strategies provide satisfactory results, it is unlikely that the quality of mappings that they produce will be comparable with manually specified mappings. The manual approach for defining mappings is a painful process, although, it can provide declarative and expressive correspondences by exploiting the knowledge of an expert for the two mapped ontologies in many different ways.
In addition to the mapping discovery, the mapping representation is a very important issue for an application that implements a mediation scenario. A set of criteria that should be taken into consideration, in order to decide which language/format should be used for the mapping representation include the following (based on [20]):

- Web compatibility
- Language independence
- Simplicity
- Expressiveness
- Purpose independence
- Executability
- Mediation task

Although, many languages (OWL [3], C-OWL [7], SWRL [22], the Alignment Format [16], MAFRA [25], EDOAL [35], [19], OMWG mapping language [36], etc.) have been proposed for the task of mapping representation, only a few combine the previous criteria. A comparison of some of these languages and formats for mapping specification is available in [20].

In this thesis, we do not focus on the discovery of the mappings between two ontologies. We are only interested in the specification and the representation of the kinds of mappings between OWL ontologies which can be exploited by a query mediation system in order to perform SPARQL query rewriting. In our knowledge, only [18], examines the problem of describing such mapping types but not directly, since it describes which mapping types cannot be used in the rewriting process. In contrast, we present in this thesis concrete mappings that can be used for SPARQL query rewriting.

### 3.2 SPARQL query rewriting

Within the Semantic Web community, the process of SPARQL query rewriting is gaining attention. It is used to perform various tasks such as query optimization, Description Logic inference, query decomposition, query translation and data integration.

SPARQL query optimization focuses on rewriting techniques that minimize the evaluation complexity using algebraic query rewriting rules and establishing relational algebra optimization techniques into the context of SPARQL ([37], [32]), or using other methodologies like selectivity estimation etc. ([5], [39], [21]).

In the field of Description Logic inference, SPARQL query rewriting is basically used for performing reasoning tasks. Currently, ontology repositories construct inference ontology models, and match SPARQL queries to the models, in order to derive inference results.
However, the size of an inference model could be prohibitive for large-scale deployment. A recent approach that performs SPARQL query rewriting using inference rules and can be used to overcome the previous problem is [23].

SPARQL query decomposition and SPARQL query translation are fundamental tasks in information integration systems. Benslimane et. al [4] proposed recently a system that performs SPARQL query decomposition in order to query distributed heterogeneous information sources. After the decomposition, the resulted SPARQL sub-queries are translated into SQL sub-queries but no algorithm or other details are provided in the paper. In contrast, Quilitz et. al [34] proposed SPARQL query decomposition, in order to overcome the large overhead in network traffic produced by the SPARQL implementations that load all the RDF graphs mentioned in a query to the local machine.

In the field of SPARQL query translation, two recent approaches [15] and [9] perform complete SPARQL query translation into SQL queries, preserving the SPARQL semantics and exploiting 1:1 cardinality mappings between an RDF data model and a relational data model. Similarly with SPARQL-to-SQL proposed methods, Bikakis et. al [6] present a framework which provides a formal mapping model for the expression of OWL to XML Schema mappings and a generic formal methodology for SPARQL-to-XQuery translation. Their methodology has the ability to exploit 1:N cardinality mappings that do not contain restrictions and composition operations, in contrast to our approach where such mapping types are supported.

Up to now, limited studies have been made in the field of query rewriting related to posing a SPARQL query over different RDF datasets. Akahani et. al [1] proposed a theoretical perspective of approximate query rewriting for submitting queries to multiple ontologies. In their approach no specific context (e.g. using SPARQL) is defined and no specific algorithms for the query rewriting process are provided.

An approach which comes closer to ours, with some of its parts based on a preliminary description of our work [26], has been presented recently by Correndo et al. [11]. They present a SPARQL query rewriting methodology for achieving RDF data mediation over linked data. Correndo et al. use transformations between RDF structures (i.e. graphs) in order to define the mappings between two ontologies. This choice seems to restrict the mappings expressivity and also the supported query types. Queries containing IRIs inside FILTER expressions cannot be handled, while the mapping definition seems to be a painful procedure. In contrast to our proposal, mappings produced by an ontology matching [20] system, need post-processing in order to assist the mapping discovery.
Chapter 4

Ontology mapping model

In order for SPARQL queries posed over a global ontology to be rewritten in terms of a local ontology, mappings between the global and local ontologies should be specified. In this chapter we present a model for the expression of mappings between OWL DL ontologies in the context of SPARQL query rewriting. In Section 4.1 we present a motivating example. In Section 4.2 and Section 4.3 we present the supported mapping types used for the query rewriting process, as well as their abstract syntax and semantics. Finally, the mapping representation is discussed in Section 4.4.

4.1 Motivating example

In this section we present a motivating example for eliciting the ontology mapping requirements of the mediator framework. Since our query rewriting methodology is generic, from this point forward we will be discussing for mappings between a source and a target ontology rather than between global and local ontologies. The mapping types presented in this section have been selected among others because they can be used for the rewriting of a SPARQL query.

Since we are working in the context of SPARQL queries, some mapping types may not be useful for the query rewriting process. For example, a mapping containing aggregates would be meaningless, since aggregates cannot be represented in the current SPARQL. Such mapping types are described in [18] and many of them could be useful for post-processing the query results but not during the query rewriting and query answering process.

In Figure 4.1, we show the structure of two different ontologies. The source ontology describes a store\(^1\) that sells various products including books and cd’s and the target ontology describes a bookstore\(^2\). The rounded corner boxes represent the classes. They

\(^{1}\)Store ontology namespace: src = http://www.ontologies.com/SourceOntology.owl#

\(^{2}\)Bookstore ontology namespace: trg = http://www.ontologies.com/TargetOntology.owl#
are followed by their properties (object and datatype). The rectangle boxes at the bottom of the figure represent individuals. The arrows represent the relationships between these basic OWL constructs.

Figure 4.1: Semantically Overlapping Ontologies.

In order to map a source ontology to a target ontology, various relationship types like equivalence (≡) and subsumption (⊆, ⊂) can be used. For example, in Figure 4.2 we present an equivalence relationship between ontology constructs and in Figure 4.3 and we present a subsumption relationship. The source ontology class Book seems to be equivalent with the target ontology class Textbook, as these two classes seem to describe individuals of the same type. Similarly, the source ontology class Product seems to subsume the target ontology class Textbook, as the class Product seems to describe various types of individuals and not only Textbook individuals.

A source ontology class can be mapped to an expression between target ontology classes. The expression may involve union (∪) and intersection (∩) operations between classes. For example, in Figure 4.4 the class Science is mapped to the union of classes ComputerScience and Mathematics, since it seems to describe both ComputerScience and Mathematics individuals. Similarly, in Figure 4.5 the class Popular is mapped to the intersection of the class BestSeller with the union of classes ComputerScience and Mathematics. This mapping emerges from the fact that the class Popular seems to describe ComputerScience and Mathematics individuals which are also of type BestSeller.
4.1. MOTIVATING EXAMPLE

Figure 4.2: A class mapping using an equivalence relationship.

Figure 4.3: A class mapping using a subsumption relationship.

Figure 4.4: A class mapping using a union operation between classes.
In addition, it is possible to restrict a class on some property values in order to form a correspondence. For example, in Figure 4.6 the class \texttt{Pocket} is mapped to the class \texttt{Textbook} restricted on its \texttt{size} property values, since the class \texttt{Pocket} seems to describe \texttt{Textbook} individuals having a specific value for the property \texttt{size} (e.g. less than or equal to 14). Similarly, in Figure 4.7 the class \texttt{Autobiography} is mapped to the class \texttt{Biography} restricted to the values of the properties \texttt{author} and \texttt{topic}. This mapping emerges from the fact that the class \texttt{Autobiography} seems to describe \texttt{Biography} individuals having the same value for these two properties.

Similarly with classes, an individual from the source ontology can be mapped to an individual from the target ontology (see Figure 4.8). In this case, only the equivalence relationship can be taken into consideration since the subsumption relationship is used mainly with sets.

Accordingly, an object/datatype property from the source ontology can be mapped to an object/datatype property from the target ontology (see Figure 4.9).
4.1. MOTIVATING EXAMPLE

Figure 4.7: A class mapping using property restrictions.

Figure 4.8: A mapping between two individuals.

Figure 4.9: A mapping between two datatype properties.
Domain and range restrictions can be useful for mappings between properties in order to restrict the individuals that participate in these two sets. For example, in Figure 4.10 the object property `partOf` from the source ontology is mapped to the object property `partOf` from the target restricted on its domain values. More specifically, the domain of the property `partOf` from the target ontology (i.e., class `Textbook`) is restricted on its `size` property values in order to match with the domain of the property `partOf` from the source ontology.

![Figure 4.10: An object property mapping using a domain restriction.](image)

In addition, an object property from the source ontology can be mapped to the inverse of an object property from the target ontology. For example, in Figure 4.11 the object property `publisher` is mapped to the inverse of the object property `publishes`, since the binary relations described by the property `publisher` correspond with the inverse binary relations described by the property `publishes`. Taking a closer look, we observe that the domain of the property `publisher` corresponds with the range of the property `publishes`, and similarly the range of the property `publisher` corresponds with the domain of the property `publishes`.

![Figure 4.11: An object property mapping using an inverse operation.](image)

Finally, a source ontology property can be mapped to an expression between target on-
4.2. ABSTRACT SYNTAX AND SEMANTICS

The basic concepts of OWL, whose mappings are useful for the rewriting process, are the classes \( c \), the object properties \( op \), the datatype properties \( dp \) and the individuals \( i \).

In order to define the mapping types which are useful for the rewriting process, we
use Description Logics (DL). We treat OWL classes as DL concepts, OWL properties as DL roles and OWL individuals as DL individuals. Following our convention, let $C, D$ be OWL classes (treated as atomic concepts), $R, S$ be OWL object properties (treated as atomic roles) and $K, L$ be OWL datatype properties (treated as atomic roles). Similarly, let $a, b, c, v_{op}$ be OWL individuals and $v_{dp}$ be a data value.

An interpretation $I$ consists of a non-empty set $\Delta^I$ (the domain of the interpretation) and an interpretation function, which assigns to every atomic concept $A$ a set $A^I \subseteq \Delta^I$, to every atomic role $B$ a binary relation $B^I \subseteq \Delta^I \times \Delta^I$ and to every individual $k$ an element $k^I \in \Delta^I$ (based on [2]).

In Tables 4.1, 4.2, 4.3, 4.4, 4.5 and 4.6 we present the set of class and property constructors which we use for the definition of mappings. In these tables we introduce some new constructors (preceded with asterisk) which should not be confused with the basic Description Logics constructors defined in [2]. In addition to the concept/role constructors, a DL knowledge base consists of assertional axioms which are presented in Table 4.7.

### Table 4.1: Class constructors used in the definition of mappings.

<table>
<thead>
<tr>
<th>Class Constructors</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Class Intersection</strong></td>
</tr>
<tr>
<td>Syntax: $C \cap D$</td>
</tr>
<tr>
<td>Semantics: $C^I \cap D^I$</td>
</tr>
<tr>
<td>Description: Creates a class that contains the instances of class $C$, which are</td>
</tr>
<tr>
<td>also instances of class $D$.</td>
</tr>
<tr>
<td><strong>Class Union</strong></td>
</tr>
<tr>
<td>Syntax: $C \cup D$</td>
</tr>
<tr>
<td>Semantics: $C^I \cup D^I$</td>
</tr>
<tr>
<td>Description: Creates a class that contains the instances of class $C$, as well as</td>
</tr>
<tr>
<td>the instances of class $D$.</td>
</tr>
<tr>
<td><em>Class Restriction - based on the value of an object property</em></td>
</tr>
<tr>
<td>Syntax: $\left( R \quad v_{op}, \quad \overline{\text{op}} \in { \neq, = } \right)$</td>
</tr>
<tr>
<td>Semantics: ${ \alpha \in C^I \mid \exists b \quad (\alpha, b) \in R^I \land b \quad \overline{\text{op}} \quad v_{op} }$</td>
</tr>
<tr>
<td>i.e. An individual $\alpha$ that belongs to the interpretation of class $C$,</td>
</tr>
<tr>
<td>belongs also to the interpretation of the constructed class, if and only if there</td>
</tr>
<tr>
<td>exists an individual $b$, such that the binary relation $(\alpha, b)$ belongs to</td>
</tr>
<tr>
<td>the interpretation of the object property $R$ and also $b$ is related to the value $v_{op}$ using a comparator $\overline{\text{op}}$.</td>
</tr>
<tr>
<td>Description: Creates a class that contains the instances of class $C$ having a</td>
</tr>
<tr>
<td>specific value (related to $v_{op}$ using a comparator $\overline{\text{op}}$) for the object property $R$.</td>
</tr>
</tbody>
</table>
Table 4.2: Class constructors used in the definition of mappings (continued from Table 4.1).

<table>
<thead>
<tr>
<th>Class Constructor</th>
<th>Syntax</th>
<th>Semantics</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Class Restriction - based on the value of a datatype property</em></td>
<td>$C.(K \ cp \ v_{dp})$, $cp \in {\neq, =, \leq, \geq, &lt;, &gt;}$</td>
<td>${\alpha \in C^I \mid \exists b. (\alpha, b) \in K^I \land b \ cp \ v_{dp}}$</td>
<td>i.e. An individual $\alpha$ that belongs to the interpretation of class $\text{C}$, belongs also to the interpretation of the constructed class, if and only if there exists a data value $b$, such that the binary relation $(\alpha, b)$ belongs to the interpretation of the datatype property $K$ and also $b$ is related to the value $v_{dp}$, using a comparator $cp$. Creates a class that contains the instances of class $C$ having a specific value (related to $v_{dp}$ using a comparator $cp$) for the datatype property $K$.</td>
</tr>
</tbody>
</table>

| *Class Restriction - based on the values of two object properties* | $C.(R \ cp \ S)$, $cp \in \{\neq, =\}$ | $\{\alpha \in C^I \mid \exists b, \exists c. (\alpha, b) \in R^I \land (\alpha, c) \in S^I \land b \ cp \ c\}$ | i.e. An individual $\alpha$ that belongs to the interpretation of class $\text{C}$, belongs also to the interpretation of the constructed class, if and only if there exists an individual $b$, as well as an individual $c$, such that the binary relation $(\alpha, b)$ belongs to the interpretation of the object property $R$, the binary relation $(\alpha, c)$ belongs to the interpretation of the object property $S$, and also $b$ is related to $c$, using a comparator $cp$. Creates a class that contains the instances of class $C$ having specific values (related to each other using a comparator $cp$) for the object properties $R, S$. |

| *Class Restriction - based on the values of two datatype properties* | $C.(K \ cp \ L)$, $cp \in \{\neq, =, \leq, \geq, <, >\}$ | $\{\alpha \in C^I \mid \exists b, \exists c. (\alpha, b) \in K^I \land (\alpha, c) \in L^I \land b \ cp \ c\}$ | i.e. An individual $\alpha$ that belongs to the interpretation of class $\text{C}$, belongs also to the interpretation of the constructed class, if and only if there exists a data value $b$, as well as a data value $c$, such that the binary relation $(\alpha, b)$ belongs to the interpretation of the datatype property $K$, the binary relation $(\alpha, c)$ belongs to the interpretation of the datatype property $L$, and also $b$ is related to $c$, using a comparator $cp$. Creates a class that contains the instances of class $C$ having specific values (related to each other using a comparator $cp$) for the datatype properties $K, L$. |
Table 4.3: Object property constructors used in the definition of mappings.

<table>
<thead>
<tr>
<th>Object Property Constructor</th>
<th>Syntax</th>
<th>Semantics</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Object Property Intersection</strong></td>
<td>$R \cap S$</td>
<td>$R^T \cap S^T$</td>
<td>Creates an object property that contains the binary relations described by the object property $R$, which are also described by the object property $S$.</td>
</tr>
<tr>
<td><strong>Object Property Union</strong></td>
<td>$R \sqcup S$</td>
<td>$R^T \cup S^T$</td>
<td>Creates an object property that contains the binary relations described by the object property $R$, as well as the binary relations described by the object property $S$.</td>
</tr>
<tr>
<td><strong>Object Property Composition</strong></td>
<td>$R \circ S$</td>
<td>${ (c, b) \mid \exists a. (a, b) \in R^T \land (b, c) \in S^T }$</td>
<td>i.e. A binary relation $(a, c)$ belongs to the interpretation of the constructed object property, if and only if there exists an individual $b$, such that the binary relation $(a, b)$ belongs to the interpretation of the object property $R$, and also $(b, c)$ belongs to the interpretation of the object property $S$.</td>
</tr>
<tr>
<td><strong>Inverse Object Property</strong></td>
<td>$\text{inv}(R)$</td>
<td>${ (b, a) \mid (a, b) \in R^T }$</td>
<td>i.e. A binary relation $(b, a)$ belongs to the interpretation of the constructed object property, if and only if there exists a binary relation $(b, c)$, which belongs to the interpretation of the object property $R$.</td>
</tr>
<tr>
<td>Description</td>
<td>Creates an object property that contains the binary relations described by the path connecting the domain of the object property $R$ to the range of the object property $S$.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 4.4: Object property constructors used in the definition of mappings (continued from Table 4.3).

<table>
<thead>
<tr>
<th>Object Property Constructors</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Object Property Restriction - based on the domain values</em></td>
</tr>
<tr>
<td><strong>Syntax</strong></td>
</tr>
<tr>
<td><strong>Semantics</strong></td>
</tr>
<tr>
<td><strong>i.e.</strong></td>
</tr>
<tr>
<td><strong>Description</strong></td>
</tr>
</tbody>
</table>

*Object Property Restriction - based on the range values*

| **Syntax** | $R.\text{range}(C)$ |
| **Semantics** | $\{(a, b) \mid (a, b) \in R^T \land b \in C_\text{I}^T\}$ |
| **i.e.** | A binary relation $(a, b)$ belongs to the interpretation of the constructed object property, if and only if the binary relation $(a, b)$ belongs also to the interpretation of the object property $R$, and the individual $b$ belongs to the interpretation of class $C$. |
| **Description** | Creates an object property that contains the binary relations described by the object property $R$, whose range values are restricted to the instances of class $C$. |

Table 4.5: Datatype property constructors used in the definition of mappings.

<table>
<thead>
<tr>
<th>Datatype Property Constructors</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Datatype Property Intersection</strong></td>
</tr>
<tr>
<td><strong>Syntax</strong></td>
</tr>
<tr>
<td><strong>Semantics</strong></td>
</tr>
<tr>
<td><strong>Description</strong></td>
</tr>
</tbody>
</table>

**Datatype Property Union**

| **Syntax** | $K \cup L$ |
| **Semantics** | $K^T \cup L^T$ |
| **Description** | Creates a datatype property that contains the binary relations described by the datatype property $K$, as well as the binary relations described by the datatype property $L$. |
Table 4.6: Datatype property constructors used in the definition of mappings (continued from Table 4.5).

<table>
<thead>
<tr>
<th>Datatype Property Composition</th>
<th>Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Datatype Property Composition</strong></td>
<td><strong>R ∘ K</strong></td>
<td>**{(α, c)</td>
</tr>
<tr>
<td>i.e. A binary relation (α, c) belongs to the interpretation of the constructed datatype property, if and only if there exists an individual b, such that the binary relation (α, b) belongs to the interpretation of the object property R, and also (b, c) belongs to the interpretation of the datatype property K.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>Creates a datatype property that contains the binary relations described by the path connecting the domain of the object property R to the range of the datatype property K.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Syntax</th>
<th><strong>K.domain(C)</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Semantics</td>
<td>**{(α, b)</td>
</tr>
<tr>
<td>i.e. A binary relation (α, b) belongs to the interpretation of the constructed datatype property, if and only if the binary relation (α, b) belongs also to the interpretation of the datatype property K, and the individual α belongs to the interpretation of class C.</td>
<td></td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>Creates a datatype property that contains the binary relations described by the datatype property K, whose domain values are restricted to the instances of class C.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Syntax</th>
<th><strong>K.range(cp, vdp), cp ∈ {≠, =, ≤, ≥, &lt;, &gt;}</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Semantics</td>
<td>**{(α, b)</td>
</tr>
<tr>
<td>i.e. A binary relation (α, b) belongs to the interpretation of the constructed datatype property, if and only if the binary relation (α, b) belongs also to the interpretation of the datatype property K, and the data value b is related to the value vdp, using a comparator cp.</td>
<td></td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>Creates a datatype property that contains the binary relations described by the datatype property K, whose range values are restricted to be related to the value vdp, using a comparator cp.</td>
</tr>
</tbody>
</table>
4.2. ABSTRACT SYNTAX AND SEMANTICS

Table 4.7: Terminological and assertional axioms used in the definition of mappings.

<table>
<thead>
<tr>
<th>Name</th>
<th>Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class inclusion</td>
<td>$C \subseteq D$</td>
<td>$C^I \subseteq D^I$</td>
</tr>
<tr>
<td></td>
<td>$C \supseteq D$</td>
<td>$C^I \supseteq D^I$</td>
</tr>
<tr>
<td>Object property inclusion</td>
<td>$R \subseteq S$</td>
<td>$R^I \subseteq S^I$</td>
</tr>
<tr>
<td></td>
<td>$R \supseteq S$</td>
<td>$R^I \supseteq S^I$</td>
</tr>
<tr>
<td>Datatype property inclusion</td>
<td>$K \subseteq L$</td>
<td>$K^I \subseteq L^I$</td>
</tr>
<tr>
<td></td>
<td>$K \supseteq L$</td>
<td>$K^I \supseteq L^I$</td>
</tr>
<tr>
<td>Class equality</td>
<td>$C \equiv D$</td>
<td>$C^I = D^I$</td>
</tr>
<tr>
<td>Object property equality</td>
<td>$R \equiv S$</td>
<td>$R^I = S^I$</td>
</tr>
<tr>
<td>Datatype property equality</td>
<td>$K \equiv L$</td>
<td>$K^I = L^I$</td>
</tr>
<tr>
<td>Individual equality</td>
<td>$a \equiv b$</td>
<td>$a^I = b^I$</td>
</tr>
</tbody>
</table>

**Definition 4.1 (Class expression).** A class expression is a class or any complex expression between two or more classes, using union or intersection operations. A class expression is denoted as $CE$ and is defined recursively in (4.1). Any class expression can be restricted to the values of one or more object property expressions $OPE$ (Definition 4.2) or datatype property expressions $DPE$ (Definition 4.3), using the comparators $\not\in \in \{\neq,=\}$ and $cp \in \{\neq,=,\leq,\geq,\leq,<,>\}$, respectively. Moreover, it is possible for a class expression to be restricted on a set of individuals having property values (either individuals $v_{op}$ or data values $v_{dp}$) with a specific relationship between them, defined either by $\not\in$ or $cp$.

$$CE := c$$
$$| CE \cap CE$$ (class intersection)
$$| CE \cup CE$$ (class union)
$$| CE.(OPE \not\in v_{op})$$ (class restricted on object property value)
$$| CE.(DPE cp v_{dp})$$ (class restricted on datatype property value)
$$| CE.(OPE_1 \not\in OPE_2)$$ (class restricted on object property values)
$$| CE.(DPE_1 cp DPE_2)$$ (class restricted on datatype property values)

**Definition 4.2 (Object property expression).** An object property expression is an object property or any complex expression between two or more object properties, using composition, union or intersection operations. An object property expression is denoted as $OPE$ and is defined recursively in (4.2). Inverse property operations are possible to appear inside an object property expression. Any object property expression can be restricted on its domain and/or range by using a class expression defining the applied restrictions.
Definition 4.3 (Datatype property expression). A datatype property expression is a datatype property or any complex expression between object and datatype properties using the composition operation, or between two or more datatype properties, using union or intersection operations. A datatype property expression is denoted as DPE and is defined recursively in (4.3). Any datatype property expression can be restricted on its domain values by using a class expression defining the applied restrictions. In addition, the range values of a datatype property expression can be restricted on various data values $v_{dp}$, using a comparator $cp \in \{\neq, =, \leq, \geq, <, >\}$.

\[
DPE := \begin{align*}
& dp \quad \text{(datatype property)} \\
| & op \circ DPE \quad \text{(datatype property composition)} \\
| & DPE \cap DPE \quad \text{(datatype property intersection)} \\
| & DPE \cup DPE \quad \text{(datatype property union)} \\
| & DPE\text{.domain}(CE) \quad \text{(datatype property restricted on domain values)} \\
| & DPE\text{.range}(CE) \quad \text{(datatype property restricted on range values)}
\end{align*}
\]
4.4. MAPPING REPRESENTATION

Class mapping. A class from a source ontology \( s \) can be mapped to a class expression from a target ontology \( t \) (shown in (4.4)).

\[
c_s \ rel \ CE_t, \ rel := \equiv | \subseteq | \supseteq
\]

Object property mapping. An object property from a source ontology \( s \) can be mapped to an object property expression from a target ontology \( t \) (shown in (4.5)).

\[
op_s \ rel \ OPE_t, \ rel := \equiv | \subseteq | \supseteq
\]

Datatype property mapping. A datatype property from a source ontology \( s \) can be mapped to a datatype property expression from a target ontology \( t \) (shown in (4.6)).

\[
dp_s \ rel \ DPE_t, \ rel := \equiv | \subseteq | \supseteq
\]

We note here that the equivalence between two different properties or between a property and a property expression, denotes equivalence between the domains and ranges of those properties or property expressions. Similarly, the subsumption relationships between two different properties or between a property and a property expression denote analogous relationships between the domains and ranges of those properties or property expressions. The proofs for the above statements are available in the Appendix B.

Individual mapping. An individual from a source ontology \( s \) can be mapped to an individual from a target ontology \( t \) (shown in (4.7)).

\[
i_s \equiv i_t
\]

4.4 Mapping representation

In the previous sections we presented the abstract syntax used for the mapping definition. Using this abstract syntax we list, in Tables 4.8, 4.9, 4.10 and 4.11, a possible set of correspondences for the ontologies presented in Figure 4.1.

In order to implement our framework, the need of a serializable language is of major importance. As mentioned in Section 3.1 many languages have been proposed for the task of mapping representation (C-OWL [7], SWRL [22], the Alignment Format [16], MAFRA [25], etc.). However, the language that fulfills the majority of our requirements is EDOAL\(^3\)

\(\text{http://alignapi.gforge.inria.fr/edoal.html}\)
Table 4.8: Class mapping examples based in Figure 4.1.

<table>
<thead>
<tr>
<th>Class Mappings</th>
</tr>
</thead>
<tbody>
<tr>
<td>a. ( src: Book \equiv trg: Textbook )</td>
</tr>
<tr>
<td>b. ( src: Product \sqsupseteq trg: Textbook )</td>
</tr>
<tr>
<td>c. ( src: Publisher \equiv trg: Publisher )</td>
</tr>
<tr>
<td>d. ( src: Collection \sqsubseteq trg: Series )</td>
</tr>
<tr>
<td>e. ( src: Novel \sqsubset trg: Literature )</td>
</tr>
<tr>
<td>f. ( src: Poetry \sqsubset trg: Literature )</td>
</tr>
<tr>
<td>g. ( src: Biography \equiv trg: Biography )</td>
</tr>
<tr>
<td>h. ( src: Autobiography \equiv trg: Biography.(trg: author = trg: topic) )</td>
</tr>
<tr>
<td>i. ( src: NewPublication \equiv trg: Computing \cap trg: NewRelease )</td>
</tr>
<tr>
<td>j. ( src: Science \equiv trg: ComputerScience \sqcup trg: Mathematics )</td>
</tr>
<tr>
<td>k. ( src: Popular \equiv (trg: ComputerScience \sqcup trg: Mathematics) \cap \sqcap trg: BestSeller )</td>
</tr>
<tr>
<td>l. ( src: Pocket \equiv trg: Textbook.(trg: size \leq 14) )</td>
</tr>
</tbody>
</table>

Table 4.9: Object property mapping examples based in Figure 4.1.

<table>
<thead>
<tr>
<th>Object Property Mappings</th>
</tr>
</thead>
<tbody>
<tr>
<td>m. ( src: publisher \equiv \text{inv}(trg: publishes) )</td>
</tr>
<tr>
<td>n. ( src: partOf \equiv trg: partOf:domain(trg: Textbook.(trg: size \leq 14)) )</td>
</tr>
</tbody>
</table>

Table 4.10: Datatype property mapping examples based in Figure 4.1.

<table>
<thead>
<tr>
<th>Datatype Property Mappings</th>
</tr>
</thead>
<tbody>
<tr>
<td>o. ( src: name \sqsupseteq trg: title )</td>
</tr>
<tr>
<td>p. ( src: id \sqsupseteq trg: isbn )</td>
</tr>
<tr>
<td>q. ( src: price \sqsupseteq trg: price )</td>
</tr>
<tr>
<td>r. ( src: review \equiv trg: editorialReview \sqcup trg: customerReview )</td>
</tr>
<tr>
<td>s. ( src: author \equiv trg: author \circ trg: name )</td>
</tr>
</tbody>
</table>

Table 4.11: Individual mapping examples based in Figure 4.1.

<table>
<thead>
<tr>
<th>Individual Mappings</th>
</tr>
</thead>
<tbody>
<tr>
<td>t. ( src: CSFoundations \equiv trg: FoundationsOfCS )</td>
</tr>
</tbody>
</table>
4.4. MAPPING REPRESENTATION

(Expressive and Declarative Ontology Alignment Language). Previous versions of this language have been defined in [19] and [35].

EDOAL combines the Alignment Format [16], which is used to represent the output of ontology matching algorithms, and the OMWG mapping language [36], which is an expressive ontology alignment language. It allows the representation of complex N:M cardinality mappings whose usage is not limited to a specific context (e.g. SPARQL query rewriting). The expressiveness, the simplicity, the Semantic Web compliance (given its RDF syntax) and the capability of using any kind of ontology language are the key features of this language.

However, in this thesis we have adapted a new version of this language, by performing minor changes in the syntax, in order for the language to match exactly the abstract syntax presented in the previous sections and also to restrict the language’s expressiveness for simplicity reasons. The general structure of this version is presented in Section 4.4.1, while a set of mapping representation examples is presented in Section 4.4.2.

4.4.1 General structure

In this section we present the general structure of the language which is used for the mapping representation. As mentioned before, the syntax presented in this section is based on the syntax of EDOAL. The default namespace applying to the constructs in the following grammar description is oml standing for http://www.music.tuc.gr/oml#, while the namespace align is equivalent to the Alignment Format namespace which is the http://knowledgeweb.semanticweb.org/heterogeneity/alignment#.

The structure of the set of mappings between two overlapping ontologies is the same as that of the Alignment Format:

```
alignment ::= <align:Alignment rdf:about="uri">
  <align:onto1> onto </align:onto1>
  <align:onto2> onto </align:onto2>
  (<align:map> cell </align:map>)*
</align:Alignment>
```

The Ontology construct contains information about an aligned ontology:

```
onto ::= <align:Ontology rdf:about="uri">
  <align:formalism> formalism </align:formalism>
</align:Ontology>
```

```
formalism ::= <align:Formalism>
  <align:uri> uri </align:uri>
  <align:name> string </align:name>
</align:Formalism>
```
The mappings between the two overlapping ontologies are structured as a set of cells:

\[
\text{cell} ::= \langle\text{align:Cell rdf:about="uri"}\
    \langle\text{align:entity1} \text{entity1} \langle/\text{align:entity1}\rangle\rangle\langle\text{align:entity2} \text{entity2} \langle/\text{align:entity2}\rangle\rangle\langle\text{align:relation} \text{relation} \langle/\text{align:relation}\rangle\rangle\langle/\text{align:Cell}\rangle
\]

The aligned construct from the source ontology can be a class, an object/datatype property, or an individual. It is worth to mention that since the current version of EDOAL provides the capability of using any kind of ontology language, it adapts a more generic vocabulary. More specifically, it refers to OWL object properties as relations, due to the fact that they relate class instances. Furthermore, it refers to OWL datatype properties as properties and to OWL individuals as instances. Finally, the term attribute is used to refer to both relations and properties. In this version, we have not made changes to the terminology which is used to describe these basic ontology constructs.

\[
\text{entity1} ::= \langle\text{Class rdf:about="uri"}/\
    | \langle\text{Relation rdf:about="uri"}/\
    | \langle\text{Property rdf:about="uri"}/\
    | \text{instance}
\]

The aligned construct from the target ontology can be a class expression, an object/datatype property expression, or an individual:

\[
\text{entity2} ::= \text{classexpr} | \text{attrexpr} | \text{instance}
\]

\[
\text{attexpr} ::= \text{propexpr} | \text{relexpr}
\]

The list of possible relationships between ontology constructs is the same with the one presented in the abstract syntax:

\[
\text{relation} ::= \text{Equivalence} | \text{Subsumes} | \text{SubsumedBy}
\]

A class expression is specified by using the construct Class. It can be either a simple class identified by its URI or a complex expression between two or more classes, using union or intersection operations. The union and intersection operations are introduced by using the constructs or and and, respectively. In all cases, the class expression can be restricted on one or more object/datatype property values, using the construct restrict.

\[
\text{classexpr} ::= \langle\text{Class rdf:about="uri"}/\
    | \langle\text{Class} \text{classconst} \langle/\text{Class}\rangle
\]
4.4. MAPPING REPRESENTATION

A class restriction on an object/datatype property value is specified by using the construct `AttributeValueRestriction`. The construct `onAttribute` represents the restricted object/datatype property, while the restricted value is represented either by the construct `instanceOrAttributeValue` or by the construct `literalValue` in accordance with various kinds of comparators (refer to the `comparator` list). The construct `instanceOrAttributeValue` may represent either an individual in order to specify a restriction on an object property value, or an object/datatype property expression in order to specify a restriction on a set of individuals having property values with a specific relationship between them. On the other hand, the construct `literalValue` represents a simple RDF literal in order to specify a restriction on a datatype property value.

```xml
<AttributeValueRestriction>
  <onAttribute> attexpr </onAttribute>
  <comparator> comparator </comparator>
  ( <instanceOrAttributeValue>
      instanceOrAttributeValue
    </instanceOrAttributeValue>
    | <literalValue> RDFLiteral </literalValue>)
</AttributeValueRestriction>
```

```xml
comparator::= Equal | NotEqual | Greater | GreaterThanOrEqual
        | Less | LessThanOrEqual
```

```xml
instanceOrAttributeValue::= instance | attexpr
```

An object property expression is specified by using the construct `Relation`. It can be either a simple object property identified by its URI or a complex expression between two or more object properties, using composition, union or intersection operations. The composition operation is introduced by using the construct `compose`. Due to the fact that inverse property operations are also possible to appear inside an object property expression, the construct `inverse` is available. In all cases, the object property expression can be restricted on its domain and/or range values, using the construct `restrict`.

```xml
relexpr::= <Relation rdf:about="uri"/>
         | <Relation> relconst </Relation>
```
According to the abstract syntax, an object property can be restricted on its domain/range values by using a class expression defining the applied restrictions. A domain or range restriction on an object property is specified by using the constructs RelationDomainRestriction and RelationRangeRestriction, respectively. The class expression which is used to restrict the domain/range values is specified by using the construct class.

A datatype property expression is specified by using the construct Property. It can be either a simple datatype property identified by its URI or any complex expression between object and datatype properties using the composition operation, or between two or more datatype properties, using union or intersection operations. In all cases, the datatype property expression can be restricted on its domain and/or range values, using the construct restrict.

A domain or range restriction on a datatype property is specified by using the constructs PropertyDomainRestriction and PropertyValueRestriction, respectively. In
a range restriction, the restricted value is represented by the construct literalValue in accordance with various kinds of comparators (refer to the comparator list). It is worth to mention that according to the abstract syntax, the range values of a datatype property expression can be restricted on data values exclusively.

\[
\text{propcond} ::= <\text{PropertyDomainRestriction}>
\quad <\text{class} \ classexpr \ </\text{class}>
\quad </\text{PropertyDomainRestriction}>
\mid <\text{PropertyValueRestriction}>
\quad <\text{comparator} \ comparator \ </\comparator>
\quad <\text{literalValue} \ RDFLiteral \ </\text{literalValue}>
\quad </\text{PropertyValueRestriction}>
\]

An individual is specified by using the construct Instance and it can be a simple URI:

\[
\text{instance} ::= <\text{Instance} rdf:about="uri"/>
\]

Finally, since previous versions of EDOAL were described as OWL DL ontologies (the current is not described), in order to provide some control to the terms of the language’s vocabulary, we have made a new version of the language’s ontology in order to match the syntax presented in this section. This new ontology version is available in the Appendix C.

### 4.4.2 Mapping representation examples

In this section we provide the representation for a set of mappings between the ontologies presented in Figure 4.1, using the syntax that we discussed in the previous subsection.

**Example 4.1.** The mapping between the class Science from the source ontology and the union of classes ComputerScience and Mathematics from the target ontology (mapping j of Table 4.8) can be represented as follows:

\[
<\text{align:Cell} rdf:about="MappingRule_j">\\
<\text{align:entity1}>\\
<\text{oml:Class} rdf:about="&src;Science"/>\\
</\text{align:entity1}>
<\text{align:entity2}>\\
<\text{oml:Class}>\\
<\text{oml:or} rdf:parseType="Collection">\\
<\text{oml:Class} rdf:about="&trg;ComputerScience"/>\\
<\text{oml:Class} rdf:about="&trg;Mathematics"/>\\
</\text{oml:or}>
</\text{oml:Class}>
</\text{align:Cell}>
\]
Example 4.2. The mapping between the object property `partOf` from the source ontology and the object property `partOf` from the target ontology (mapping n of Table 4.9) which is restricted on its domain values can be represented as follows:

```xml
<oml:Relation rdf:about="&src;partOf"/>
<oml:restrict rdf:parseType="Collection">
  <oml:Relation rdf:about="&trg;partOf"/>
  <oml:RelationDomainRestriction>
    <oml:class>
      <oml:restrict rdf:parseType="Collection">
        <oml:Class rdf:about="&trg;Textbook"/>
        <oml:AttributeValueRestriction>
          <oml:onAttribute>
            <oml:Property rdf:about="&trg;size"/>
          </oml:onAttribute>
          <oml:comparator>LessThanOrEqual</oml:comparator>
          <oml:literalValue rdf:datatype="&xsd;int">14</oml:literalValue>
        </oml:AttributeValueRestriction>
      </oml:Class>
    </oml:restrict>
  </oml:RelationDomainRestriction>
</oml:Relation>
```
Example 4.3. The mapping between the datatype property author from the source ontology and the composition of the object property author with the datatype property name from the target ontology (mapping s of Table 4.10) can be represented as follows:

\[
\text{Example 4.3. The mapping between the datatype property author from the source ontology and the composition of the object property author with the datatype property name from the target ontology (mapping s of Table 4.10) can be represented as follows:}
\]

\[
<\text{align:Cell rdf:about="MappingRule_s">}
<\text{align:entity1}>
<\text{oml:Property rdf:about="&src;author"}>
</\text{align:entity1}>
<\text{align:entity2}>
<\text{oml:Property}>
<\text{oml:compose rdf:parseType="Collection"}>
<\text{oml:Relation rdf:about="&trg;author"}>
<\text{oml:Property rdf:about="&trg;name"}>
</\text{oml:compose}>
</\text{oml:Property}>
</\text{align:entity2}>
<\text{align:relation}Equivalence</\text{align:relation}>
</\text{align:Cell}>
\]

Example 4.4. The mapping between the individual CSFoundations from the source ontology and the individual FoundationsOfCS from the target ontology (mapping t of Table 4.11) can be represented as follows:

\[
\text{Example 4.4. The mapping between the individual CSFoundations from the source ontology and the individual FoundationsOfCS from the target ontology (mapping t of Table 4.11) can be represented as follows:}
\]

\[
<\text{align:Cell rdf:about="MappingRule_t">}
<\text{align:entity1}>
<\text{oml:Instance rdf:about="&src;CSFoundations"}>
</\text{align:entity1}>
<\text{align:entity2}>
<\text{oml:Instance rdf:about="&trg;FoundationsOfCS"}>
</\text{align:entity2}>
<\text{align:relation}Equivalence</\text{align:relation}>
</\text{align:Cell}>
\]

A more comprehensive example, showing the representation of the complete set of mappings defined in Tables 4.8, 4.9, 4.10 and 4.11, is available in the Appendix D.
Chapter 5

SPARQL query rewriting overview

In this chapter we present an overview of the SPARQL query rewriting process. Query rewriting is done by exploiting a predefined set of mappings which is based on the different mapping types described in Section 4.3.

The SPARQL query rewriting process lies in the query’s graph pattern rewriting. The rewritten query is produced by replacing the rewritten graph pattern to the initial query’s graph pattern. Consequently the rewriting process is independent of the query type (i.e. SELECT, CONSTRUCT, ASK, DESCRIBE) and the SPARQL solution sequence modifiers (i.e. ORDER BY, DISTINCT, REDUCED, LIMIT, OFFSET).

Since a graph pattern consists basically of triple patterns, the most important part of a SPARQL query rewriting is the triple pattern rewriting. Triple patterns may refer to data (e.g. relationships between instances) or schema (e.g. relationships between classes and/or properties) information, or to both. In order to present the rewriting procedure in depth and due to the inability of handling all the different triple pattern types in the same manner, we distinguish triple patterns into Data Triple Patterns (see Definition 5.1) and Schema Triple Patterns (see Definition 5.2).

We consider triple patterns of the form \((\text{subject}, \text{predicate}, \text{object})\) as defined before. Let \(L\) be the set of literals, \(V\) the set of variables, \(I\) the set of IRIs, \(I_{\text{RDF}}\) the set containing the IRIs of the RDF vocabulary (e.g. \(\text{rdf} : \text{type}\)), \(I_{\text{RDFS}}\) the set containing the IRIs of the RDF Schema vocabulary (e.g. \(\text{rdfs} : \text{subClassOf}\)) and \(I_{\text{OWL}}\) the set containing the IRIs of the OWL vocabulary (e.g. \(\text{owl} : \text{equivalentClass}\)).

**Definition 5.1 (Data Triple Pattern).** The triple patterns that only apply to data and not schema info are considered to be Data Triple Patterns. A tuple \(t \in DTP\) (Data Triple Pattern set - shown in (5.1)) is a Data Triple Pattern.
\[ DTP = (I' \cup L \cup V) \times (I' \cup \{\text{rdf : type, owl : sameAs}\}) \times (I' \cup L \cup V) \] (5.1)

\[ I' = I - \text{I}_{\text{RDF}} - \text{I}_{\text{RDFS}} - \text{I}_{\text{OWL}} \] (5.2)

**Definition 5.2 (Schema Triple Pattern).** The triple patterns that only apply to schema and not data info are considered to be Schema Triple Patterns. A tuple \( t \in \text{STP} \) (Schema Triple Pattern set - shown in (5.3)) is a Schema Triple Pattern.

\[ \text{STP} = ((I \cup L \cup V) \times I \times (I \cup L \cup V)) - DTP \] (5.3)

The factor which is mainly used for the categorization of a triple pattern is the triple pattern’s predicate part. The only exception occurs when the predicate part of a triple pattern contains the RDF property \( \text{rdf : type} \). In this case, the object part of the triple pattern should be checked. If the triple pattern’s object part contains an RDF/RDFS/OWL IRI, then the triple pattern concerns schema info (e.g. \( (?x, \text{rdf : type, owl : Class}) \)). Otherwise, if the triple pattern’s object part contains another type of IRI, then the triple pattern concerns data info (e.g. \( (?x, \text{rdf : type, src : Product}) \)). In Table 5.1 we present the categorization of a triple pattern set, into Data/Schema Triple Patterns. Triple patterns having a variable on their predicate part are not taken into consideration, since they may apply either to data or schema info.

<table>
<thead>
<tr>
<th>Category</th>
<th>Triple Pattern ((s, p, o))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Triple Patterns</td>
<td>((?x, \text{rdf : type, src : Product}))  ((?x, \text{src : author, ?y}))  ((?x, \text{src : price, &quot;12&quot;^^xsd : int}))</td>
</tr>
<tr>
<td>Schema Triple Patterns</td>
<td>((?x, \text{rdfs : subClassOf, src : Product}))  ((\text{src : author, rdfs : domain, ?x}))  ((\text{src : Pocket, owl : equivalentClass, ?x}))</td>
</tr>
<tr>
<td>Non-categorized (non-supported)</td>
<td>((\text{src : CSFoundations, ?x, &quot;52&quot;^^xsd : int}))  ((\text{src : Popular, ?x, src : Science}))</td>
</tr>
</tbody>
</table>

Since a triple pattern consists of three parts (subject, predicate, object), in order to rewrite it we have to follow a three-step procedure by exploiting mappings for each triple part.
pattern’s part. Firstly, a triple pattern is rewritten using the mapping which has been specified for its predicate part, resulting to a graph pattern which may contain one or more triple patterns. Then, the resulted graph pattern is rewritten triple pattern by triple pattern, using the mappings of the triple patterns’ object parts. Finally, the same procedure is repeated for the triple patterns’ subject parts. It is worth to mention that SPARQL variables, blank nodes, literal constants and RDF/RDFS/OWL IRIs which may appear in the subject, predicate or object of a triple pattern remain the same during the rewriting procedure. Consequently, the SPARQL variables of the initial query appear also in the rewritten query.

In Chapter 6, we provide a set of functions that perform Data Triple Pattern rewriting using predefined mappings for a triple pattern’s subject, predicate and object parts. Similarly, in Chapter 7 we provide the functions that perform Schema Triple Pattern rewriting.

In addition to the triple patterns, a graph pattern may contain filters. The SPARQL variables, literal constants, operators ($\&\&$, $||$, $!$, $!=$, $>$, $>=$, $<=$, $+,$ $-$, $*=\) and built-in functions (e.g. bound, isIRI, isLiteral, datatype, lang, str, regex) which may appear inside a FILTER expression remain the same during the rewriting process. For class IRIs and property IRIs which may appear inside a FILTER expression of a SPARQL query, we use 1:1 cardinality mappings for the expression rewriting.

We note that the rewriting of a triple pattern, is not dependent on mapping relationships (i.e. equivalence, subsumption). These relationships, affect only the evaluation results of the rewritten query over the target ontology. A complete algorithm and a set of examples, that show the graph pattern rewriting process, are presented in Chapter 8.
Chapter 6

Data Triple Pattern rewriting

In this chapter, we provide the functions that perform Data Triple Pattern rewriting. These functions are actually rewriting steps in the process of Data Triple Pattern rewriting and are also semantics preserving (see Definition 6.1). They provide the rewritten form of a Data Triple Pattern using a mapping for its subject, predicate or object part, and they are based on the mapping type. In Table 6.1, we present the notation used for the definition of these functions.

Definition 6.1 (Semantics preserving rewriting). Let $DS_s$ be the RDF dataset of a source ontology, and let $DS_t$ be the RDF dataset of a target ontology. Similarly, let $DS_m$ be the RDF dataset which is produced by merging the $DS_s$ and $DS_t$ datasets using a set of mappings $M$.

Given a complete set (i.e. a set that contains every possible mapping) of sound (i.e. valid) mappings $M$ between $DS_s$ and $DS_t$, the rewriting step performed for a triple pattern $t$, based on a mapping $\mu \in M$, is semantics preserving if and only if the evaluation result of $t$ and the evaluation result of the rewritten graph pattern $gp'$ over $DS_m$, preserve the mapping semantics.

In other words, having a set $J = \text{var}(t)$ of SPARQL variables, the relationship ($\equiv$, $\subseteq$, $\supseteq$) that holds for the mappings used in the rewriting process, should also hold between $[[t]]_{DS_m}$ and $[[gp']]_{DS_m}$ projected on $J$. Refer to Section 2.3.5 for the notation, as well as for the SPARQL graph pattern semantics.

$$[[t]]_{DS_m} \text{ rel } \pi^*_J ([[gp']]_{DS_m}), \text{ rel:} = \equiv | \subseteq | \supseteq$$ (6.1)

$$J = \text{var}(t) \cap \text{var}(gp') = \text{var}(t)$$ (6.2)
Table 6.1: The notation used for the Data Triple Pattern rewriting functions.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_s$</td>
<td>The subscript $s$ denotes that the entity $x$ (class, object property, datatype property or individual) belongs to the source ontology.</td>
</tr>
<tr>
<td>$x_t$</td>
<td>The subscript $t$ denotes that the entity $x$ (class, object property, datatype property or individual) belongs to the target ontology.</td>
</tr>
<tr>
<td>$D^x_y(t, \mu)$</td>
<td>The $D$ function takes two arguments: a Data Triple Pattern $t$ and a mapping $\mu$ for the subject, predicate or object part of $t$. This function is used to provide the resulted form of $t$, after being rewritten based on its subject, predicate or object part, using the mapping $\mu$. The triple pattern’s part which is used for the rewriting is denoted by the superscript $x \in {s, p, o}$. The subscript $y \in {c, op, dp, i, *}$ shows the type of $x$ (e.g. class, object property, etc.). The asterisk denotes any type.</td>
</tr>
</tbody>
</table>

In Section 6.1 we describe the Data Triple Pattern rewriting process using a mapping for the triple pattern’s subject part, while in Sections 6.2 and 6.3 we present the Data Triple Pattern rewriting process using mappings for the triple pattern’s object and predicate parts, respectively.

### 6.1 Rewriting based on triple pattern’s subject part

Generally, when a class or a property appears on the subject part of a triple pattern we conclude that the triple pattern involves schema info, as there is no way for a non RDF/RDFS/OWL IRI to appear at the same time in the triple pattern’s predicate part. Thus, the only case mentioned for the rewriting of a Data Triple Pattern by its subject part concerns individuals.

**Rewriting based on individual mapping.** Let $i_s$ be an individual from the source ontology which is mapped to an individual $i_t$ from the target ontology. Having a Data Triple Pattern $t = (i_s, predicate, object)$ with $i_s$ in its subject part and anything in its predicate and object parts, we can rewrite it by its subject part, using a predefined mapping $\mu$ and the function (6.3).

$$D^s_y(t, \mu) = (i, predicate, object) \text{ if } \mu : i_s \equiv i_t$$

**Example 6.1.** Consider the query posed over the source ontology of Figure 4.1: “Return the type of the CSFoundations individual”. The SPARQL syntax of the source query is shown below:
6.2. REWRITING BASED ON TRIPLE PATTERN’S OBJECT PART

@PREFIX src: <http://www.ontologies.com/SourceOntology.owl#>.
@PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.

SELECT ?x
WHERE {src:CSFoundations rdf:type ?x.}

In order to rewrite the SPARQL query for posing it over the target ontology of Figure 4.1, we have to rewrite the triple pattern \( t = (\text{src} : \text{CSFoundations}, \text{rdf} : \text{type}, ?x) \) by its subject, predicate and object parts. Taking into consideration a mapping \( \mu \) of the triple pattern’s subject part, the result of the triple pattern’s \( t \) rewriting by its subject part is provided by invoking the function (6.3).

\[
t = (\text{src} : \text{CSFoundations}, \text{rdf} : \text{type}, ?x)
\]

\[
\mu : \text{src} : \text{CSFoundations} \equiv \text{trg} : \text{FoundationsOfCS}
\]

Using the parameters defined above, as well as the function (6.3), the triple pattern \( t \) is rewritten as follows:

\[
\mathcal{D}_i^s(t, \mu) = (\text{trg} : \text{FoundationsOfCS}, \text{rdf} : \text{type}, ?x)
\]

In Lemma 6.1 we summarize the functions presented in this section, which are used for the rewriting of a Data Triple Pattern based on a mapping for the triple pattern’s subject part.

**Lemma 6.1.** Let \( i_s \) be an individual from the source ontology. Having a Data Triple Pattern \( t \) and a predefined mapping \( \mu \) for its subject part, we can rewrite it by its subject, by invoking the function (6.4). Considering the semantics of the initial triple pattern, as well as the semantics of the resulted graph pattern, this rewriting step is semantics preserving.

\[
\mathcal{D}_i^s(t, \mu) = \begin{cases} 
\mathcal{D}_i^s(t, \mu) & \text{if } t = (i_s, \text{predicate}, \text{object}) \\
\emptyset & \text{elsewhere} 
\end{cases}
\]  

(6.4)

The proof of Lemma 6.1 is available in the Appendix B.

## 6.2 Rewriting based on triple pattern’s object part

When a property appears on the object part of a triple pattern, we conclude that the triple pattern deals with schema info, as there is no way for a non RDF/RDFS/OWL IRI to appear at the same time in the triple pattern’s predicate part. Similarly, in case that
a class appears on a triple pattern’s object part, the only factor which can be used to
determine the triple pattern’s type (Data or Schema Triple Pattern), is whether the RDF
property rdf:type appears on the predicate part or not. Thus, the only cases mentioned
for the rewriting of a Data Triple Pattern by its object part concern individuals, as well
as classes with the precondition that the RDF property rdf:type appears on the triple
pattern’s predicate part at the same time.

Rewriting based on class mapping. Let \( c_s \) be a class from the source ontology which
is mapped to a class expression from the target ontology. Having a Data Triple Pattern
\( t = (\text{subject}, \text{rdf:type}, c_s) \) with the class \( c_s \) in its object part, the RDF property rdf:type
in its predicate and anything in its subject part, we can rewrite it by its object part, using
a predefined mapping \( \mu \) and the function (6.5), which is continued in (6.6).

\[
\begin{align*}
D^\mu_c(t, \mu) = \begin{cases} 
(s\text{ubject}, \text{rdf:type}, c_t) & \text{if } \mu : c_s \rightarrow c_t \\
D^\mu_c(t_1, \mu_1) \cup D^\mu_c(t_2, \mu_2) & \text{if } \mu : c_s \rightarrow c_{t_1} \cup c_{t_2}, \\
& \text{where } t_1 = (\text{subject}, \text{rdf:type}, c_{t_1}), \\
& \mu_1 : c_{t_1} \equiv CE_{t_1}, \\
& \text{and } t_2 = (\text{subject}, \text{rdf:type}, c_{t_2}), \\
& \mu_2 : c_{t_2} \equiv CE_{t_2} \\
D^\mu_c(t_1, \mu_1) \cap D^\mu_c(t_2, \mu_2) & \text{if } \mu : c_s \rightarrow c_{t_1} \cap c_{t_2}, \\
& \text{where } t_1 = (\text{subject}, \text{rdf:type}, c_{t_1}), \\
& \mu_1 : c_{t_1} \equiv CE_{t_1}, \\
& \text{and } t_2 = (\text{subject}, \text{rdf:type}, c_{t_2}), \\
& \mu_2 : c_{t_2} \equiv CE_{t_2} \\
D^\mu_c(t_1, \mu_1) \cap D^\mu_{op}(t_2, \mu_2) & \text{if } \mu : c_s \rightarrow c_{t_1}.(o\text{p}, \text{v}_{op}), \\
& \text{where } \text{v}_{op} \in \{\neq, =\}, \\
& \text{v}_{op} = \text{individual}, \\
& t_1 = (\text{subject}, \text{rdf:type}, c_t), \\
& \mu_1 : c_t \equiv CE_t, \\
& \text{and } t_2 = (\text{subject}, \text{op}_t, \text{v}_{op}), \\
& \mu_2 : \text{op}_t \equiv OPE_t \\
\end{cases}
\end{align*}
\]

*continued in (6.6)
6.2. REWRITING BASED ON TRIPLE PATTERN’S OBJECT PART

Do\(c(t)\) =

\[\begin{align*}
&D_p^e(t_1, \mu_1) \text{ AND } D_p^o(t_2, \mu_2) \text{ AND } D_p^d_p(t_3, \mu_3) \\
&\text{FILTER(?var1 cP ?var2)}
end{align*}\]

\[D_c^e(t, \mu) =\]

\[\begin{align*}
&D_p^e(t_1, \mu_1) \text{ AND } D_p^o(t_2, \mu_2) \text{ AND } D_p^d_p(t_3, \mu_3) \\
&\text{FILTER(?var1 cP ?var2)}
end{align*}\]

\[\begin{align*}
&D_p^e(t_1, \mu_1) \text{ AND } D_p^o(t_2, \mu_2) \text{ AND } D_p^d_p(t_3, \mu_3) \\
&\text{FILTER(?var1 cP ?var2)}
end{align*}\]

continued from (6.5)

The functions \(D_p^o\) and \(D_p^d_p\) are used by the function (6.5) in order to provide the graph pattern that forms a restricted property and are defined in Section 6.3.

**Example 6.2.** Consider the query posed over the source ontology of Figure 4.1: “Return the poetry books”. The SPARQL syntax of the source query is shown below:

```sparql
@PREFIX src: <http://www.ontologies.com/SourceOntology.owl#>.
@PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.

SELECT ?x
WHERE {
}
```

In order to rewrite the SPARQL query for posing it over the target ontology of Figure 4.1, we have to rewrite the triple pattern \(t = (?x, rdf:type, src:Poetry)\) by its
subject, predicate and object parts. Taking into consideration a mapping \( \mu \) of the triple pattern’s object part, the result of the triple pattern’s \( t \) rewriting by its object part is provided by invoking the function (6.5).

\[
t = (\?x, \text{rdf}\text{:type}, \text{src}\text{:Poetry})
\]

\[
\mu : \text{src}\text{:Poetry} \sqsubseteq \text{trg}\text{:Literature}
\]

The mapping \( \mu \) is of type \( c_s \to c_t \). Thus, using the parameters defined above, as well as the function (6.5), the triple pattern \( t \) is rewritten as follows:

\[
D^\mu_c(t, \mu) = (\?x, \text{rdf}\text{:type}, \text{trg}\text{:Literature})
\]

**Example 6.3.** Consider the query posed over the source ontology of Figure 4.1: “Return the scientific books”. The SPARQL syntax of the source query is shown below:

```sparql
@PREFIX src: <http://www.ontologies.com/SourceOntology.owl#>.
@PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.

SELECT ?x
WHERE {?x \text{rdf}\text{:type} \text{src}\text{:Science}.}
```

In order to rewrite the SPARQL query for posing it over the target ontology of Figure 4.1, we have to rewrite the triple pattern \( t = (\?x, \text{rdf}\text{:type}, \text{src}\text{:Science}) \) by its subject, predicate and object parts. Taking into consideration a mapping \( \mu \) of the triple pattern’s object part, the result of the triple pattern’s \( t \) rewriting by its object part is provided by invoking the function (6.5).

\[
t = (\?x, \text{rdf}\text{:type}, \text{src}\text{:Science})
\]

\[
\mu : \text{src}\text{:Science} \equiv \text{trg}\text{:ComputerScience} \sqcup \text{trg}\text{:Mathematics}
\]

The mapping \( \mu \) is of type \( c_s \to c_t_1 \sqcup c_t_2 \). Following the definition of the function (6.5), two triple patterns \( t_1 \) and \( t_2 \) are created and the complex mapping \( \mu \) is decomposed into the mappings \( \mu_1 \) and \( \mu_2 \). The triple patterns \( t_1 \) and \( t_2 \) contain the classes \( c_t_1 \) and \( c_t_2 \) on their object part, respectively. The mapping of the class \( c_t_1 \) is provided by \( \mu_1 \), while the mapping of the class \( c_t_2 \) is provided by \( \mu_2 \).

\[
t_1 = (\?x, \text{rdf}\text{:type}, c_t_1)
\]

\[
t_2 = (\?x, \text{rdf}\text{:type}, c_t_2)
\]

\[
\mu_1 : c_t_1 \equiv \text{trg}\text{:ComputerScience}
\]

\[
\mu_2 : c_t_2 \equiv \text{trg}\text{:Mathematics}
\]
Thus,

\[ D_c^o(t, \mu) = D_c^o(t_1, \mu_1) \cup D_c^o(t_2, \mu_2) \]

The mappings \( \mu_1 \) and \( \mu_2 \) are of type \( c_s \rightarrow c_t \). Thus, using the parameters defined above, as well as the function (6.5) for the rewriting of the triple patterns \( t_1 \) and \( t_2 \), the initial triple pattern \( t \) is rewritten as follows:

\[ D_c^o(t, \mu) = D_c^o(t_1, \mu_1) \cup D_c^o(t_2, \mu_2) \]

\[ = (\exists x. \text{rdf} : \text{type}, \text{trg} : \text{ComputerScience}) \cup \]

\[ (\exists x. \text{rdf} : \text{type}, \text{trg} : \text{Mathematics}) \]

**Example 6.4.** Consider the query posed over the source ontology of Figure 4.1: “Return the popular scientific books”. The SPARQL syntax of the source query is shown below:

@PREFIX src: <http://www.ontologies.com/SourceOntology.owl#>.
@PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.

SELECT ?x WHERE {?x rdf:type src:Popular.}

In order to rewrite the SPARQL query for posing it over the target ontology of Figure 4.1, we have to rewrite the triple pattern \( t = (\exists x. \text{rdf} : \text{type}, \text{src} : \text{Popular}) \) by its subject, predicate and object parts. Taking into consideration a mapping \( \mu \) of the triple pattern’s object part, the result of the triple pattern’s \( t \) rewriting by its object part is provided by invoking the function (6.5).

\[ t = (\exists x. \text{rdf} : \text{type}, \text{src} : \text{Popular}) \]

\[ \mu : \text{src} : \text{Popular} \equiv (\text{trg} : \text{ComputerScience} \sqcup \text{trg} : \text{Mathematics}) \sqcap \text{trg} : \text{BestSeller} \]

The mapping \( \mu \) is of type \( c_s \rightarrow c_t \cap c_t \). Following the definition of the function (6.5), two triple patterns \( t_1 \) and \( t_2 \) are created and the complex mapping \( \mu \) is decomposed into the mappings \( \mu_1 \) and \( \mu_2 \). The triple patterns \( t_1 \) and \( t_2 \) contain the classes \( c_{t1} \) and \( c_{t2} \) on their object part, respectively. The mapping of the class \( c_{t1} \) is provided by \( \mu_1 \), while the mapping of the class \( c_{t2} \) is provided by \( \mu_2 \).

\[ t_1 = (\exists x. \text{rdf} : \text{type}, c_{t1}) \]

\[ t_2 = (\exists x. \text{rdf} : \text{type}, c_{t2}) \]
$\mu_1 : c_{t_1} \equiv \text{trg:ComputerScience} \sqcup \text{trg:Mathematics}$

$\mu_2 : c_{t_2} \equiv \text{trg:BestSeller}$

Thus,

$$D^o_c(t, \mu) = D^o_c(t_1, \mu_1) \text{ AND } D^o_c(t_2, \mu_2)$$

Similarly, the resulted complex mapping $\mu_1$ is of type $c_s \rightarrow c_t$. Consequently, two triple patterns $t_3$ and $t_4$ are created and the complex mapping $\mu_1$ is decomposed into the mappings $\mu_3$ and $\mu_4$. The triple patterns $t_3$ and $t_4$ contain the classes $c_{t_3}$ and $c_{t_4}$ on their object part, respectively. The mapping of the class $c_{t_3}$ is provided by $\mu_3$, while the mapping of the class $c_{t_4}$ is provided by $\mu_4$.

$$t_3 = (?x, \text{rdf:type}, c_{t_3})$$

$$t_4 = (?x, \text{rdf:type}, c_{t_4})$$

$\mu_3 : c_{t_3} \equiv \text{trg:ComputerScience}$

$\mu_4 : c_{t_4} \equiv \text{trg:Mathematics}$

Thus,

$$D^o_c(t, \mu) = D^o_c(t_1, \mu_1) \text{ AND } D^o_c(t_2, \mu_2)$$

$$= (D^o_c(t_3, \mu_3) \text{ UNION } D^o_c(t_4, \mu_4)) \text{ AND } D^o_c(t_2, \mu_2)$$

The mappings $\mu_2$, $\mu_3$ and $\mu_4$ are of type $c_s \rightarrow c_t$. Thus, using the parameters defined above, as well as the function (6.5) for the rewriting of the triple patterns $t_2$, $t_3$ and $t_4$, the initial triple pattern $t$ is rewritten as follows:

$$D^o_c(t, \mu) = D^o_c(t_1, \mu_1) \text{ AND } D^o_c(t_2, \mu_2)$$

$$= (D^o_c(t_3, \mu_3) \text{ UNION } D^o_c(t_4, \mu_4)) \text{ AND } D^o_c(t_2, \mu_2)$$

$$= (?x, \text{rdf:type, trg:ComputerScience}) \text{ UNION }$$

$$($$?x, \text{rdf:type, trg:Mathematics}) \text{ AND }$$

$$($$?x, \text{rdf:type, trg:BestSeller})$$
Rewriting based on individual mapping. Let \( i_s \) be an individual from the source ontology which is mapped to an individual \( i_t \) from the target ontology. Having a Data Triple Pattern \( t = (\text{subject}, \text{predicate}, i_s) \) with \( i_s \) in its object part and anything in its predicate and subject parts, we can rewrite it by its object part, using a predefined mapping \( \mu \) and the function (6.7).

\[
D^t_i(t, \mu) = (\text{subject}, \text{predicate}, i_t) \quad \text{if} \quad i_s \equiv i_t
\]  
(6.7)

Example 6.5. Consider the query posed over the source ontology of Figure 4.1: “Return the individuals which are specified to be the same with the CSFoundations individual”. The SPARQL syntax of the source query is shown below:

```sparql
@PREFIX src: <http://www.ontologies.com/SourceOntology.owl#>.
@PREFIX owl: <http://www.w3.org/2002/07/owl#>.
SELECT ?x
WHERE {
?x owl:sameAs src:CSFoundations.
}
```

In order to rewrite the SPARQL query for posing it over the target ontology of Figure 4.1, we have to rewrite the triple pattern \( t = (?x, \text{owl:sameAs}, \text{src:CSFoundations}) \) by its subject, predicate and object parts. Taking into consideration a mapping \( \mu \) of the triple pattern’s object part, the result of the triple pattern’s \( t \) rewriting by its object part is provided by invoking the function (6.7).

\[
t = (?x, \text{owl:sameAs}, \text{src:CSFoundations})
\]
\[
\mu : \text{src:CSFoundations} \equiv \text{trg:FoundationsOfCS}
\]

Using the parameters defined above, as well as the function (6.7), the triple pattern \( t \) is rewritten as follows:

\[
D^t_i(t, \mu) = (?x, \text{owl:sameAs}, \text{trg:FoundationsOfCS})
\]
In Lemma 6.2 we summarize the functions presented in this section, which are used for the rewriting of a Data Triple Pattern based on a mapping for the triple pattern’s object part.

**Lemma 6.2.** Let $i_s$ be an individual and $c_s$ be a class from the source ontology. Having a Data Triple Pattern $t$ and a predefined mapping $\mu$ for its object part, we can rewrite it by its object, by invoking the function (6.8). Considering the semantics of the initial triple pattern, as well as the semantics of the resulted graph pattern, this rewriting step is semantics preserving.

\[
D^r_o(t, \mu) = \begin{cases} 
D^r_o(t, \mu) & \text{if } t = (\text{subject, predicate, } i_s) \\
D^r_o(t, \mu) & \text{if } t = (\text{subject, rdf:type, } c_s) \\
\emptyset & \text{elsewhere} 
\end{cases}
\]  \hspace{1cm} (6.8)

The proof of Lemma 6.2 is available in the Appendix B.

\[\square\]

### 6.3 Rewriting based on triple pattern’s predicate part

In order to rewrite a Data Triple Pattern by its predicate part only property mappings can be used, since a class or an individual cannot appear on a triple pattern’s predicate part.

**Rewriting based on object property mapping.** Let $op_s$ be an object property from the source ontology which is mapped to an object property expression from the target ontology. Having a Data Triple Pattern $t = (\text{subject, } op_s, \text{object})$ with $op_s$ in its predicate part and anything in its subject and object parts, we can rewrite it by its predicate part, using a predefined mapping $\mu$ and the function (6.9).
6.3. REWRITING BASED ON TRIPLE PATTERN’S PREDICATE PART

\[
(\text{subject}, op_1, object) \quad \text{if } \mu : op_s \rightarrow op_t \\
D_{op}^p(t_1, \mu_1) \land D_{op}^p(t_2, \mu_2) \quad \text{if } \mu : op_s \rightarrow op_1 \circ op_2,
\]
where \(t_1 = (\text{subject}, op_1, \text{?var})\),
\(\mu_1 : op_1 \equiv \text{OPE}_1\),
and \(t_2 = (\text{?var}, op_2, \text{object})\),
\(\mu_2 : op_2 \equiv \text{OPE}_2\)

\[
D_{op}^p(t_1, \mu_1) \lor D_{op}^p(t_2, \mu_2) \quad \text{if } \mu : op_s \rightarrow op_1 \sqcap op_2,
\]
where \(t_1 = (\text{subject}, op_1, \text{object})\),
\(\mu_1 : op_1 \equiv \text{OPE}_1\),
and \(t_2 = (\text{subject}, op_2, \text{object})\),
\(\mu_2 : op_2 \equiv \text{OPE}_2\)

\[
D_{op}^p(t_1, \mu_1) \land D_{op}^p(t_2, \mu_2) \quad \text{if } \mu : op_s \rightarrow \text{inv}(op_t),
\]
where \(t_1 = (\text{object}, op_1, \text{subject})\)
and \(\mu_1 : op_t \equiv \text{OPE}_t\)

\[
D_{op}^p(t_1, \mu_1) \land D_{op}^p(t_2, \mu_2) \quad \text{if } \mu : op_s \rightarrow \text{dom}(c_t),
\]
where \(t_1 = (\text{subject}, op_1, \text{object})\),
\(\mu_1 : op_1 \equiv \text{OPE}_t\),
and \(t_2 = (\text{subject}, \text{rdf} : \text{type}, c_t)\),
\(\mu_2 : c_t \equiv \text{CET}\)

\[
D_{op}^p(t_1, \mu_1) \land D_{op}^p(t_2, \mu_2) \quad \text{if } \mu : op_s \rightarrow \text{ran}(c_t),
\]
where \(t_1 = (\text{subject}, op_1, \text{object})\),
\(\mu_1 : op_1 \equiv \text{OPE}_t\),
and \(t_2 = (\text{object}, \text{rdf} : \text{type}, c_t)\),
\(\mu_2 : c_t \equiv \text{CET}\)

Example 6.6. Consider the query posed over the source ontology of Figure 4.1: “Return the publisher of the book CSFoundations”. The SPARQL syntax of the source query is shown below:
SELECT ?x
WHERE {src:CSFoundations src:publisher ?x.}

In order to rewrite the SPARQL query for posing it over the target ontology of Figure 4.1, we have to rewrite the triple pattern \( t = (\text{src:CSFoundations}, \text{src:publisher}, ?x) \) by its subject, predicate and object parts. Taking into consideration a mapping \( \mu \) of the triple pattern’s predicate part, the result of the triple pattern’s \( t \) rewriting by its predicate part is provided by invoking the function (6.9).

\[
t = (\text{src:CSFoundations}, \text{src:publisher}, ?x)
\]

\[
\mu : \text{src:publisher} \equiv \text{inv( trg:publishes )}
\]

The mapping \( \mu \) is of type \( \text{op}_s \rightarrow \text{inv( op_t )} \). Following the definition of the function (6.9), a triple patterns \( t_1 \) is created and the complex mapping \( \mu \) is transformed to the mapping \( \mu_1 \). The triple pattern \( t_1 \) contains an object property \( \text{op}_t \) on its predicate part and its mapping is provided by \( \mu_1 \).

\[
t_1 = (?x, \text{op}_t, \text{src:CSFoundations})
\]

\[
\mu_1 : \text{op}_t \equiv \text{trg:publishes}
\]

Thus,

\[
D_{\text{op}}^p(t, \mu) = D_{\text{op}}^p(t_1, \mu_1)
\]

The mapping \( \mu_1 \) is of type \( \text{op}_s \rightarrow \text{op}_t \). Thus, using the parameters defined above, as well as the function (6.9) for the rewriting of the triple pattern \( t_1 \), the initial triple pattern \( t \) is rewritten as follows:

\[
D_{\text{op}}^p(t, \mu) = D_{\text{op}}^p(t_1, \mu_1)
\]

\[
= (?x, \text{trg:publishes}, \text{src:CSFoundations})
\]

**Rewriting based on datatype property mapping.** Let \( dp_s \) be a datatype property from the source ontology which is mapped to a datatype property expression from the target ontology. Having a Data Triple Pattern \( t = (\text{subject}, dp_s, \text{object}) \) with \( dp_s \) in its predicate part and anything in its subject and object parts, we can rewrite it by its predicate part, using a predefined mapping \( \mu \) and the function (6.10).
Example 6.7. Consider the query posed over the source ontology of Figure 4.1: “Return the name of the CSFoundations individual which is of type Book”. The SPARQL syntax of the source query is shown below:

@PREFIX src: <http://www.ontologies.com/SourceOntology.owl#>.

SELECT ?x
WHERE {src:CSFoundations src:name ?x.}
In order to rewrite the SPARQL query for posing it over the target ontology of Figure 4.1, we have to rewrite the triple pattern \( t = (\text{src} : \text{CSFoundations}; \text{src} : \text{name}; ?x) \) by its subject, predicate and object parts. Taking into consideration a mapping \( \mu \) of the triple pattern’s predicate part, the result of the triple pattern’s \( t \) rewriting by its predicate part is provided by invoking the function (6.10).

\[
t = (\text{src} : \text{CSFoundations}, \text{src} : \text{name}, ?x)
\]

\[
\mu : \text{src} : \text{name} \supseteq \text{trg} : \text{title}
\]

The mappings \( \mu \) is of type \( dp_s \rightarrow dp_t \). Thus, using the parameters defined above, as well as the function (6.10), the triple pattern \( t \) is rewritten as follows:

\[
D^p_{dp}(t, \mu) = (\text{src} : \text{CSFoundations}, \text{trg} : \text{title}, ?x)
\]

**Example 6.8.** Consider the query posed over the source ontology of Figure 4.1: “Return the available reviews for the book \textit{CSFoundations}”. The SPARQL syntax of the source query is shown below:

\[
\text{PREFIX src: <http://www.ontologies.com/SourceOntology.owl#>.
SELECT ?x
WHERE {src:CSFoundations src:review ?x.}
\]

In order to rewrite the SPARQL query for posing it over the target ontology of Figure 4.1, we have to rewrite the triple pattern \( t = (\text{src} : \text{CSFoundations}, \text{src} : \text{review}; ?x) \) by its subject, predicate and object parts. Taking into consideration a mapping \( \mu \) of the triple pattern’s predicate part, the result of the triple pattern’s \( t \) rewriting by its predicate part is provided by invoking the function (6.10).

\[
t = (\text{src} : \text{CSFoundations}, \text{src} : \text{review}, ?x)
\]

\[
\mu : \text{src} : \text{review} \equiv \text{trg} : \text{editorialReview} \sqcup \text{trg} : \text{customerReview}
\]

The mapping \( \mu \) is of type \( dp_s \rightarrow dp_t \sqcup dp_{t2} \). Following the definition of the function (6.10), two triple patterns \( t_1 \) and \( t_2 \) are created and the complex mapping \( \mu \) is decomposed into the mappings \( \mu_1 \) and \( \mu_2 \). The triple patterns \( t_1 \) and \( t_2 \) contain the datatype properties \( dp_{t1} \) and \( dp_{t2} \) on their predicate part, respectively. The mapping of the datatype property \( dp_{t1} \) is provided by \( \mu_1 \), while the mapping of the datatype property \( dp_{t2} \) is provided by \( \mu_2 \).

\[
t_1 = (\text{src} : \text{CSFoundations}, dp_{t1}, ?x)
\]

\[
t_2 = (\text{src} : \text{CSFoundations}, dp_{t2}, ?x)
\]
6.4. COMBINATION EXAMPLES

\[ t_2 = (\text{src} : \text{CSFoundations}, \text{dp}_{t2}, ?x) \]

\[ \mu_1 : \text{dp}_{t1} \equiv \text{trg} : \text{editorialReview} \]

\[ \mu_2 : \text{dp}_{t2} \equiv \text{trg} : \text{customerReview} \]

Thus,

\[ D_{dp}^p(t, \mu) = D_{dp}^p(t_1, \mu_1) \cup D_{dp}^p(t_2, \mu_2) \]

The mappings \( \mu_1 \) and \( \mu_2 \) are of type \( \text{dp}_{s} \rightarrow \text{dp}_{t} \). Thus, using the parameters defined above, as well as the function (6.10) for the rewriting of the triple patterns \( t_1 \) and \( t_2 \), the initial triple pattern \( t \) is rewritten as follows:

\[ D_{dp}^p(t, \mu) = D_{dp}^p(t_1, \mu_1) \cup D_{dp}^p(t_2, \mu_2) \]

\[ = (\text{src} : \text{CSFoundations}, \text{trg} : \text{editorialReview}, ?x) \cup (\text{src} : \text{CSFoundations}, \text{trg} : \text{customerReview}, ?x) \]

In Lemma 6.3 we summarize the functions presented in this section, which are used for the rewriting of a Data Triple Pattern based on a mapping for the triple pattern’s predicate part.

**Lemma 6.3.** Let \( \text{op}_s \) be an object property and \( \text{dp}_s \) be a datatype property from the source ontology. Having a Data Triple Pattern \( t \) and a predefined mapping \( \mu \) for its predicate part, we can rewrite it by its predicate, by invoking the function (6.11). Considering the semantics of the initial triple pattern, as well as the semantics of the resulted graph pattern, this rewriting step is semantics preserving.

\[
D_p^p(t, \mu) = \begin{cases} 
D_{dp}^p(t, \mu) & \text{if } t = (\text{subject}, \text{op}_s, \text{object}) \\
D_{dp}^p(t, \mu) & \text{if } t = (\text{subject}, \text{dp}_s, \text{object}) \\
\emptyset & \text{elsewhere}
\end{cases} \quad (6.11)
\]

The proof of Lemma 6.3 is available in the Appendix B.

6.4 Combination examples

In this section we provide a set of examples that combine some of the functions presented in the previous sections in order to rewrite a triple pattern based on a specific triple pattern’s part (i.e. subject, predicate, object).
Example 6.9. Consider the query posed over the source ontology of Figure 4.1: “Return the pocket-sized books”. The SPARQL syntax of the source query is shown below:

@PREFIX src: <http://www.ontologies.com/SourceOntology.owl#>.
@PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.

SELECT ?x
WHERE { ?x rdf:type src:Pocket. }

In order to rewrite the SPARQL query for posing it over the target ontology of Figure 4.1, we have to rewrite the triple pattern \( t = (?x, \text{rdf} : \text{type}, \text{src} : \text{Pocket}) \) by its subject, predicate and object parts. Taking into consideration a mapping \( \mu \) of the triple pattern’s object part, the result of the triple pattern’s \( t \) rewriting by its object part is provided by invoking the function (6.5).

\[
t = (?x, \text{rdf} : \text{type}, \text{src} : \text{Pocket}) \quad \mu : \text{src} : \text{Pocket} \equiv \text{trg} : \text{Textbook} \quad \text{(trg} : \text{size} \leq 14)
\]

Taking a closer look at the mapping \( \mu \), we conclude that the source ontology’s class \texttt{Pocket} is mapped to the target ontology’s class \texttt{Textbook}, restricted on its \texttt{size} property values. Consequently, the mapping \( \mu \) is of type \( c_s \rightarrow c_t \cap (d_{pt} \cap v_{dp}) \). Following the definition of the function (6.5), two triple patterns \( t_1 \) and \( t_2 \) are created and the complex mapping \( \mu \) is decomposed into the mappings \( \mu_1 \) and \( \mu_2 \). The triple pattern \( t_1 \) contains a class \( c_t \) on its object part, while the triple pattern \( t_2 \) contains a datatype property \( dp_t \) on its predicate part. The mapping of the class \( c_t \) is provided by \( \mu_1 \), while the mapping of the property \( dp_t \) is provided by \( \mu_2 \).

\[
t_1 = (?x, \text{rdf} : \text{type}, c_t) \\
t_2 = (?x, dp_t, \text{?var}) \\
\mu_1 : c_t \equiv \text{trg} : \text{Textbook} \\
\mu_2 : dp_t \equiv \text{trg} : \text{size}
\]

Thus,

\[
\mathcal{D}_c^o(t, \mu) = \mathcal{D}_c^o(t_1, \mu_1) \text{ AND } \mathcal{D}_{dp_t}^p(t_2, \mu_2) \text{ FILTER(\text{?var} \leq 14)}
\]

The mapping \( \mu_2 \), as well as the triple pattern \( t_2 \) are used by the function (6.10), in order to form the graph pattern representing the \texttt{size} property. Thus, using the parameters defined above, as well as the function (6.5) for the rewriting of the triple pattern \( t_1 \), the initial triple pattern \( t \) is rewritten as follows:
Example 6.10. Consider the query posed over the source ontology of Figure 4.1: “Return the autobiography books”. The SPARQL syntax of the source query is shown below:

```sparql
@PREFIX src: <http://www.ontologies.com/SourceOntology.owl#>.
@PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
SELECT ?x
WHERE { ?x rdf:type src:Autobiography. }
```

In order to rewrite the SPARQL query for posing it over the target ontology of Figure 4.1, we have to rewrite the triple pattern \( t = (?)x, rdf : type, src : Autobiography \) by its subject, predicate and object parts. Taking into consideration a mapping \( \mu \) of the triple pattern’s object part, the result of the triple pattern’s \( t \) rewriting by its object part is provided by invoking the function (6.5).

\[
\begin{align*}
D_C(t, \mu) &= D_C^o(t_1, \mu_1) \text{ AND } D_{dp}^o(t_2, \mu_2) \text{ FILTER}(?var \leq 14) \\
&= (?x, rdf : type, trg : Textbook) \text{ AND } (?x, trg : size, ?var) \\
&\text{FILTER}(?var \leq 14)
\end{align*}
\]

Taking a closer look at the mapping \( \mu \), we conclude that the source ontology’s class Autobiography is mapped to the target ontology’s class Biography, restricted on its author property values. The mapping \( \mu \) is of type \( c_s \rightarrow c_t . (dp_{t1} \text{ cp } dp_{t2}) \). Following the definition of the function (6.5), three triple patterns \( t_1, t_2 \) and \( t_3 \) are created and the complex mapping \( \mu \) is decomposed into the mappings \( \mu_1, \mu_2 \) and \( \mu_3 \). The triple pattern \( t_1 \) contains a class \( c_t \) on its object part, while the triple patterns \( t_2 \) and \( t_3 \) contain the datatype properties \( dp_{t1} \) and \( dp_{t2} \) on their predicate part, respectively. The mapping of the class \( c_t \) is provided by \( \mu_1 \), the mapping of the property \( dp_{t1} \) is provided by \( \mu_2 \) and the mapping of the property \( dp_{t2} \) is provided by \( \mu_3 \).

\[
\begin{align*}
t_1 &= (?)x, rdf : type, c_t \\
t_2 &= (?)x, dp_{t1}, ?var_1 \\
t_3 &= (?)x, dp_{t2}, ?var_2 \\
\mu_1 : c_t &\equiv trg : Biography
\end{align*}
\]
Thus,

$$D_c^\mu(t, \mu) = D_c^{\mu_1}(t_1, \mu_1) \ \mbox{AND} \ D_{dp}^{\mu_2}(t_2, \mu_2) \ \mbox{AND} \ D_{dp}^{\mu_3}(t_3, \mu_3) \ \mbox{FILTER}(\var_1 = \var_2)$$

Example 6.11. Consider the query posed over the source ontology of Figure 4.1: “Return the pocket-sized books which are part of a collection”. The SPARQL syntax of the source query is shown below:

@PREFIX src: <http://www.ontologies.com/SourceOntology.owl#>.

SELECT ?x ?y
WHERE {?x src:partOf ?y.}

In order to rewrite the SPARQL query for posing it over the target ontology of Figure 4.1, we have to rewrite the triple pattern $$t = (?x, src : partOf, ?y)$$ by its subject, predicate and object parts. Taking into consideration a mapping $$\mu$$ of the triple pattern’s predicate part, the result of the triple pattern’s $$t$$ rewriting by its predicate part is provided by invoking the function (6.9).

$$t = (?x, src : partOf, ?y)$$

$$\mu : src : partOf \equiv trg : partOf.domain(trg : Textbook.(trg : size \leq 14))$$
6.4. COMBINATION EXAMPLES

Taking a closer look at the mapping μ, we conclude that the source ontology’s property `partOf` is mapped to the target ontology’s property `partOf`, restricted on its domain values. Consequently, the mapping μ is of type $op_s \rightarrow op_t.domain(c_{t1})$. Following the definition of the function (6.9), two triple patterns $t_1$ and $t_2$ are created and the complex mapping μ is decomposed into the mappings $μ_1$ and $μ_2$. The triple pattern $t_1$ contains an object property $op_t$ on its predicate part and its mapping is provided by $μ_1$, while the triple pattern $t_2$ contains a class $c_{t1}$ on its object part and its mapping is provided by $μ_2$.

$$t_1 = (?x, op_t, ?y)$$

$$t_2 = (?x, rdf: type, c_{t1})$$

$$μ_1: op_t \equiv trg: partOf$$

$$μ_2: c_{t1} \equiv trg: Textbook.(trg: size \leq 14)$$

Thus,

$$\mathcal{D}_{op}(t, μ) = \mathcal{D}_{op}(t, μ_1) \text{ AND } \mathcal{D}_{c}(t_2, μ_2)$$

The mapping $μ_2$ actually specifies a mapping between the domain of the source ontology’s property `partOf` and the target ontology’s class `Textbook`, restricted on its `size` property values. Following the definition of the function (6.5), two triple patterns $t_3$ and $t_4$ are created and the complex mapping $μ_2$ is decomposed into the mappings $μ_3$ and $μ_4$. The triple pattern $t_3$ contains a class $c_{t2}$ on its object part, while the triple pattern $t_4$ contains a datatype property $dp_t$ on its predicate part. The mapping of the class $c_{t2}$ is provided by $μ_3$, while the mapping of the property $dp_t$ is provided by $μ_4$.

$$t_3 = (?x, rdf: type, c_{t2})$$

$$t_4 = (?x, dp_t, ?var)$$

$$μ_3: c_{t2} \equiv trg: Textbook$$

$$μ_4: dp_t \equiv trg: size$$

Thus,

$$\mathcal{D}_{op}(t, μ) = \mathcal{D}_{op}(t, μ_1) \text{ AND } \mathcal{D}_{c}(t_2, μ_2)$$

$$= \mathcal{D}_{op}(t_1, μ_1) \text{ AND } (\mathcal{D}_{c}(t_3, μ_3) \text{ AND } \mathcal{D}_{dp}(t_4, μ_4) \text{ FILTER(?var \leq 14)})$$
The mapping $\mu_4$, as well as the triple pattern $t_4$ are used by the function (6.10) in order to form the graph pattern representing the size property. Thus, using the parameters defined above, as well as the functions (6.9) and (6.5) for the rewriting of the triple patterns $t_1$ and $t_3$, the initial triple pattern $t$ is rewritten as follows:

$$D_{op}(t, \mu) = D_{op}(t_1, \mu_1) \text{ AND } D_{op}(t_2, \mu_2)$$

$$= D_{op}(t_1, \mu_1) \text{ AND } (D_c(t_3, \mu_3) \text{ AND } D_{dp}(t_4, \mu_4) \text{ FILTER(?var \leq 14)})$$

$$= (?x, trg : partOf, ?y) \text{ AND } (?x, rdf : type, trg : Textbook) \text{ AND } (?x, trg : size, ?var) \text{ FILTER(?var \leq 14)})$$

**Example 6.12.** Consider the query posed over the source ontology of Figure 4.1: “Return the authors of the book CSFoundations”. The SPARQL syntax of the source query is shown below:

```sparql
@PREFIX src: <http://www.ontologies.com/SourceOntology.owl#>.
SELECT ?x
WHERE {src:CSFoundations src:author ?x}.
```

In order to rewrite the SPARQL query for posing it over the target ontology of Figure 4.1, we have to rewrite the triple pattern $t = (src : CSFoundations, src : author, ?x)$ by its subject, predicate and object parts. Taking into consideration a mapping $\mu$ of the triple pattern’s predicate part, the result of the triple pattern’s $t$ rewriting by its predicate part is provided by invoking the function (6.10).

$$t = (src : CSFoundations, src : author, ?x)$$

$$\mu : src : author \equiv trg : author \circ trg : name$$

The mapping $\mu$ is of type $dp_s \rightarrow op_t \circ dp_t$. Following the definition of the function (6.10), two triple patterns $t_1$ and $t_2$ are created and the complex mapping $\mu$ is decomposed into the mappings $\mu_1$ and $\mu_2$. The triple patterns $t_1$ and $t_2$ contain an object property $op_t$ and a datatype property $dp_t$ on their predicate part, respectively. The mapping of the object property $op_t$ is provided by $\mu_1$, while the mapping of the datatype property $dp_t$ is provided by $\mu_2$.

$$t_1 = (src : CSFoundations, op_t, ?var)$$
6.4. COMBINATION EXAMPLES

\[ t_2 = (?var, dp_1, ?x) \]

\[ \mu_1 : op_t \equiv trg : author \]

\[ \mu_2 : dp_t \equiv trg : name \]

Thus,

\[ \mathcal{D}_{dp}(t, \mu) = \mathcal{D}_{op}(t_1, \mu_1) \text{ AND } \mathcal{D}_{dp}(t_2, \mu_2) \]

The mapping \( \mu_1 \) is of type \( op_s \rightarrow op_t \), while the mapping \( \mu_2 \) is of type \( dp_s \rightarrow dp_t \). Thus, using the parameters defined above, as well as the functions (6.9) and (6.10) for the rewriting of the triple patterns \( t_1 \) and \( t_2 \), the initial triple pattern \( t \) is rewritten as follows:

\[ \mathcal{D}_{dp}(t, \mu) = \mathcal{D}_{op}(t_1, \mu_1) \text{ AND } \mathcal{D}_{dp}(t_2, \mu_2) \]

\[ = (src : CSFoundations, trg : author, ?var) \text{ AND } (\?var, trg : name, ?x) \]
Chapter 7

Schema Triple Pattern rewriting

In order to rewrite a triple pattern any mapping type presented in Section 4.3 can be used. However, in some cases the mapped expressions should be relaxed in order for a mapping to be used by the Schema Triple pattern rewriting process.

In the Example 7.1, we show that a mapping is used as it is in order to be exploited by the Data Triple Pattern rewriting process. However, the same mapping should be relaxed in order to be used for the rewriting of a Schema Triple Pattern.

Example 7.1. In Figure 4.1, let the source ontology’s class Pocket be mapped as equivalent to the class Textbook from the target ontology, restricted on its size property values. This correspondence can be represented as follows:

\[ \mu : \ src : \text{Pocket} \equiv \ trg : \text{Textbook}.(\text{trg} : \text{size} \leq 14) \]

Having a Data Triple Pattern \( t = (?x, \text{rdf} : \text{type}, src : \text{Pocket}) \) and the mapping \( \mu \), it is clear that the entire mapping should be used to rewrite the triple pattern \( t \). This results from the fact that the mapping \( \mu \) relates a class from the source ontology with an unnamed class (i.e. set of instances) from the target ontology, and the triple pattern \( t \) concerns data info and specifically a set of instances.

On the contrary, having the mapping \( \mu \) that we presented before, as well as a Schema Triple Pattern \( t' = (src : \text{Pocket}, \text{rdf} : \text{subClassOf}, ?x) \), it is clear that the class restriction cannot be used to rewrite it. As mentioned before, the class Pocket is mapped to an unnamed class. Thus, using the class restriction for the rewriting of \( t' \), makes the evaluation results prone to whether the target ontology defines the unnamed class, which is very unlikely, and contains schema info about it as well. Consequently, in order to rewrite the triple pattern \( t' \) and also to avoid tricky hypotheses, the mapping \( \mu \) should
be transformed to a similar one, having the property restriction of the target ontology’s mapped expression removed (i.e. \( \mu' : \text{src} : \text{Pocket} \subseteq \text{trg} : \text{Textbook} \)). Such a relaxation step, seems to be reliable for Schema Triple Patterns, in the sense that it is based on some inferred facts which are more likely to return the desirable query results.

The operations that determine whether a mapping should be relaxed in order to be used for the rewriting of a Schema Triple Pattern are the following:

- Class expression restrictions.
- Object/datatype property expression restrictions on domain/range values.
- Composition operations between object/datatype property expressions.
- Inverse object property expression operations.

Mapped expressions containing the above operations are relaxed in order to be used for the rewriting of a Schema Triple Pattern. In this case, a mapped class expression \( CE \) (see Definition 4.1) is transformed to a similar class expression \( CE' \) (defined recursively in (7.1)), having any class restrictions removed.

\[
CE' := c \quad \text{(class)}
| CE' \cap CE' \quad \text{(class intersection)}
| CE' \cup CE' \quad \text{(class union)}
\]  

(7.1)

A mapped object property expression \( OPE \) (see Definition 4.2) is transformed to a similar object property expression \( OPE' \) (defined recursively in (7.2)), having any domain/range restrictions, any composed object property expressions and any inverse object property expressions removed.

\[
OPE' := op \quad \text{(object property)}
| OPE' \cap OPE' \quad \text{(object property intersection)}
| OPE' \cup OPE' \quad \text{(object property union)}
\]  

(7.2)

Similarly, a mapped datatype property expression \( DPE \) (see Definition 4.3) is transformed to a similar datatype property expression \( DPE' \) (defined recursively in (7.3)), having any domain/range restrictions and any composed property expressions removed.

\[
DPE' := dp \quad \text{(datatype property)}
| DPE' \cap DPE' \quad \text{(datatype property intersection)}
| DPE' \cup DPE' \quad \text{(datatype property union)}
\]  

(7.3)

Mappings containing mapped expressions that need relaxation, are transformed by substituting the mapped expression with the relaxed one and by modifying the mapping’s relationship, respectively. The relaxation operations presented above, can also exclude a
mapping from being used for the rewriting of a Schema Triple Pattern. For example, a
mapping between an object property and a composition of object properties is excluded,
since the relaxation method will remove the composition operation and consequently the
entire mapped expression. It is worth to say that mappings between individuals do no
need any relaxation in order to be used for the rewriting of a Schema Triple Pattern.

Even after preprocessing the defined mappings, a Schema Triple Pattern should be
rewritten differently compared to a Data Triple Pattern. The need for handling differently
these two different triple pattern types lies on the fact that a Data Triple Pattern deals
with data info (e.g. relationships between instances or between instances and data values),
while a Schema Triple Pattern deals with schema info (e.g. hierarchies and relationships
between named classes and/or named properties).

In the Example 7.2, we show that handling the rewriting of a Schema Triple Pattern
in the same manner with a Data Triple Pattern does not preserve the mapping semantics.

Example 7.2. In Figure 4.1, let the source ontology’s class Science be mapped as equivalent to the union of classes ComputerScience and Mathematics from the target ontology.
This correspondence can be represented as follows:

\[ \mu : \text{src:Science} \equiv \text{trg:ComputerScience} \cup \text{trg:Mathematics} \]

A Data Triple Pattern \( t = (\text{?x, rdf:type, src:Science}) \), involves the instances of
class Science. Taking into consideration the mapping \( \mu \), the rewritten graph pattern of \( t \)
should return the instances of the class ComputerScience, as well as the instances of the
class Mathematics, using the UNION graph pattern operator.

On the contrary, a Schema Triple Pattern \( t' = (\text{src:Science, rdf:subClassOf, ?x}) \)
involves the superclasses of the class Science. Using the mapping \( \mu \) in order to rewrite
\( t' \), someone would expect the rewritten graph pattern to return the superclasses of the
union of classes ComputerScience and Mathematics. However, such a rewritten graph
pattern is very unlikely to match any RDF graph (i.e. no results obtained), due to the
fact that the union of classes ComputerScience and Mathematics is not a named class
in the target ontology, in order to contain schema info about it. In addition, this differs
from returning the superclasses of the class ComputerScience, as well as the superclasses
of the class Mathematics, following the treatment which was used for Data Triple Pattern
rewriting.

One method to make a rewritten graph pattern semantically correspondent to the
initial triple pattern \( t' \) is by using inference. In this case, a graph pattern that matches
the common superclasses of the classes ComputerScience and Mathematics forms the
solution.

In order to rewrite a Schema Triple Pattern using 1:N cardinality mappings, simple
types of inference based on DL axioms are used. The Schema Triple Patterns which can be
handled using inference are those having on their predicate part one of the OWL/RDF/RDFS properties appearing on the set $SSP$ (Supported Schema Predicates - see (7.4)).

$$SSP = \left\{ \begin{array}{l}
  rdf : type, \\
  rdfs : subClassOf, \\
  rdfs : subPropertyOf, \\
  owl : equivalentClass, \\
  owl : equivalentProperty, \\
  owl : complementOf, \\
  owl : disjointWith
\end{array} \right\}$$  \hspace{1cm} (7.4)

Let $SSP_c$ (see (7.5)) be the supported OWL/RDF/RDFS property set which can be applied on classes, and $SSP_p$ (see (7.6)) be the supported OWL/RDF/RDFS property set which can be applied on properties.

$$SSP_c = \left\{ \begin{array}{l}
  rdf : type, \\
  rdfs : subClassOf, \\
  owl : equivalentClass, \\
  owl : complementOf, \\
  owl : disjointWith
\end{array} \right\}$$  \hspace{1cm} (7.5)

$$SSP_p = \left\{ \begin{array}{l}
  rdf : type, \\
  rdfs : subPropertyOf, \\
  owl : equivalentProperty
\end{array} \right\}$$  \hspace{1cm} (7.6)

The sets presented above are divided further, to the sets $SSP'_c$ (see (7.7)) and $SSP'_p$ (see (7.8)), respectively, for the purpose of common inference treatment.

$$SSP'_c = SSP_c - \{ rdfs : subClassOf \}$$  \hspace{1cm} (7.7)

$$SSP'_p = SSP_p - \{ rdfs : subPropertyOf \}$$  \hspace{1cm} (7.8)

Let $B, C, D, G$ be atomic concepts (i.e. classes) and $K, L, R, S$ be atomic roles (i.e. properties). Table 7.1 and Table 7.2 summarize the class and property axioms which are used for the rewriting of Schema Triple Patterns, respectively. We note that the complement operation is denoted by using the superscript $c$. 
Table 7.1: Class axioms used for the rewriting of Schema Triple Patterns.

<table>
<thead>
<tr>
<th>Type</th>
<th>Axioms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subsumption</td>
<td>if ( B \sqsubseteq C ) and ( G \equiv C ) then ( B \sqsubseteq G )</td>
</tr>
<tr>
<td></td>
<td>if ( B \sqsubseteq C ) and ( B \sqsubseteq D ) and ( G \equiv C \cap D ) then ( B \sqsubseteq G )</td>
</tr>
<tr>
<td></td>
<td>if ( B \sqsupset C ) or ( B \sqsubseteq D ) and ( G \equiv C \cup D ) then ( B \sqsubseteq G )</td>
</tr>
<tr>
<td></td>
<td>if ( B \sqsupset C ) and ( G \equiv C ) then ( B \nsubseteq G )</td>
</tr>
<tr>
<td></td>
<td>if ( B \nsubseteq C ) or ( B \nsubseteq D ) and ( G \equiv C \nsubseteq D ) then ( B \nsubseteq G )</td>
</tr>
<tr>
<td>Equivalence</td>
<td>if ( B \equiv C ) and ( G \equiv C ) then ( B \equiv G )</td>
</tr>
<tr>
<td></td>
<td>if ( B \equiv C ) and ( B \equiv D ) and ( G \equiv C \cap D ) then ( B \equiv G )</td>
</tr>
<tr>
<td>Complementarity</td>
<td>if ( B \equiv C^c ) and ( G \equiv C ) then ( B \equiv G^c )</td>
</tr>
<tr>
<td></td>
<td>if ( B \equiv C^c ) and ( B \equiv D^c ) and ( G \equiv C \cap D ) then ( B \equiv G^c )</td>
</tr>
<tr>
<td></td>
<td>if ( B \equiv C^c ) and ( B \equiv D^c ) and ( G \equiv C \cup D ) then ( B \equiv G^c )</td>
</tr>
<tr>
<td>Disjointness</td>
<td>if ( B \cap C = \emptyset ) and ( G \equiv C ) then ( B \cap G = \emptyset )</td>
</tr>
<tr>
<td></td>
<td>if ( B \cap C = \emptyset ) and ( B \cap D = \emptyset ) and ( G \equiv C \cap D ) then ( B \cap G = \emptyset )</td>
</tr>
<tr>
<td></td>
<td>if ( B \cap C = \emptyset ) and ( B \cap D = \emptyset ) and ( G \equiv C \cup D ) then ( B \cap G = \emptyset )</td>
</tr>
</tbody>
</table>

Table 7.2: Property axioms used for the rewriting of Schema Triple Patterns.

<table>
<thead>
<tr>
<th>Type</th>
<th>Axioms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subsumption</td>
<td>if ( K \sqsubseteq L ) and ( S \equiv L ) then ( K \sqsubseteq S )</td>
</tr>
<tr>
<td></td>
<td>if ( K \sqsubseteq L ) and ( K \sqsubseteq R ) and ( S \equiv L \cap R ) then ( K \sqsubseteq S )</td>
</tr>
<tr>
<td></td>
<td>if ( K \sqsubseteq L ) or ( K \sqsubseteq R ) and ( S \equiv L \cup R ) then ( K \sqsubseteq S )</td>
</tr>
<tr>
<td></td>
<td>if ( K \nsubseteq L ) and ( S \equiv L ) then ( K \nsubseteq S )</td>
</tr>
<tr>
<td></td>
<td>if ( K \nsubseteq L ) or ( K \nsubseteq R ) and ( S \equiv L \cap R ) then ( K \nsubseteq S )</td>
</tr>
<tr>
<td></td>
<td>if ( K \nsubseteq L ) and ( K \nsubseteq R ) and ( S \equiv L \cup R ) then ( K \nsubseteq S )</td>
</tr>
<tr>
<td>Equivalence</td>
<td>if ( K \equiv L ) and ( S \equiv L ) then ( K \equiv S )</td>
</tr>
<tr>
<td></td>
<td>if ( K \equiv L ) and ( K \equiv R ) and ( S \equiv L \cap R ) then ( K \equiv S )</td>
</tr>
<tr>
<td></td>
<td>if ( K \equiv L ) and ( K \equiv R ) and ( S \equiv L \cup R ) then ( K \equiv S )</td>
</tr>
</tbody>
</table>
It is worth to say that in case of 1:1 cardinality mappings, every Schema Triple Pattern having any OWL/RDF/RDFS property on its predicate part can be rewritten. In the following sections, we provide the functions that perform Schema Triple Pattern rewriting. They provide the rewritten form of a Schema Triple Pattern using a mapping for its subject, predicate or object part, and they are based on the mapping type. In Table 7.3, we present the notation used for the definition of these functions.

Table 7.3: The notation used for the Schema Triple Pattern rewriting functions.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_{s}$</td>
<td>The subscript $s$ denotes that the entity $x$ (class, object property, datatype property or individual) belongs to the source ontology.</td>
</tr>
<tr>
<td>$x_{t}$</td>
<td>The subscript $t$ denotes that the entity $x$ (class, object property, datatype property or individual) belongs to the target ontology.</td>
</tr>
<tr>
<td>$S_{y}^{x}(t, \mu)$</td>
<td>The $S$ function takes two arguments: a Schema Triple Pattern $t$ and a mapping $\mu$ for the subject, predicate or object part of $t$. This function is used to provide the resulted form of $t$ after being rewritten based on its subject, predicate or object part, using the mapping $\mu$. The triple pattern’s part which is used for the rewriting is denoted by the superscript $x \in {s, p, o}$. The subscript $y \in {c, op, dp, i, *}$ shows the type of $x$ (e.g. class, object property, etc.). The asterisk denotes any type.</td>
</tr>
</tbody>
</table>

In Section 7.1 we describe the Schema Triple Pattern rewriting process using a mapping for the triple pattern’s subject part, while in Section 7.2 we present the Schema Triple Pattern rewriting process using a mapping for the triple pattern’s object part. The rewriting of a Schema Triple Pattern by its predicate part does not result in modifications since the triple pattern’s predicate part is an RDF/RDFS/OWL property and does not affect the rewriting procedure.

### 7.1 Rewriting based on triple pattern’s subject part

A class or a property may appear in the subject part of a Schema Triple Pattern, as opposed to the Data Triple Patterns.

**Rewriting based on class mapping.** Let $c_s$ be a class from the source ontology which is mapped to a class expression from the target ontology. Having a Schema Triple Pattern $t = (c_s, predicate, object)$ with $c_s$ in its subject part, an RDF/RDFS/OWL property in its predicate and anything in its object part, we can rewrite it by its subject part, using a predefined mapping $\mu$ and the function $(7.9)$. 
Example 7.3. Consider the query posed over the source ontology of Figure 4.1: “Return the superclasses of the class Science”. The SPARQL syntax of the source query is shown below:

@PREFIX src: <http://www.ontologies.com/SourceOntology.owl#>.
@PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>.

SELECT ?x
WHERE {src:Science rdfs:subClassOf ?x.}

In order to rewrite the SPARQL query for posing it over the target ontology of Figure 4.1, we have to rewrite the triple pattern \( t = (\text{src:Science}, \text{rdfs:subClassOf}, ?x) \) by its subject, predicate and object parts. Taking into consideration a mapping \( \mu \) of the triple pattern’s subject part, the result of the triple pattern’s \( t \) rewriting by its subject part is provided by invoking the function (7.9).

\[
\begin{align*}
S'_{t}(t, \mu) &= \begin{cases} 
(c_1, \text{predicate}, \text{object}) & \text{if } \mu : c_1 \to c_1 \\
S'_{t_1}(t_1, \mu_1) \cup S'_{t_2}(t_2, \mu_2) & \text{if } \mu : c_1 \to c_1 \cap c_2 \text{ and } \\
& \text{predicate} = \text{rdfs:subClassOf}, \\
& \text{where } t_1 = (c_{t_1}, \text{predicate}, \text{object}), \\
& \mu_1 : c_{t_1} \equiv CE_{c_1}, \\
& \text{and } t_2 = (c_{t_2}, \text{predicate}, \text{object}), \\
& \mu_2 : c_{t_2} \equiv CE_{c_2} \\
S'_{t_1}(t_1, \mu_1) \land S'_{t_2}(t_2, \mu_2) & \text{if } \mu : c_1 \to c_1 \mid \cup c_2 \text{ and } \\
& \text{predicate} \in \text{SSP}, \\
& \text{or if } \mu : c_1 \to c_1 \cup c_2 \text{ and } \\
& \text{predicate} = \text{rdfs:subClassOf}, \\
& \text{where } t_1 = (c_{t_1}, \text{predicate}, \text{object}), \\
& \mu_1 : c_{t_1} \equiv CE_{c_1}, \\
& \text{and } t_2 = (c_{t_2}, \text{predicate}, \text{object}), \\
& \mu_2 : c_{t_2} \equiv CE_{c_2}
\end{cases}
\end{align*}
\]
the mappings $\mu_1$ and $\mu_2$. The triple patterns $t_1$ and $t_2$ contain the classes $c_{t1}$ and $c_{t2}$ on their subject part, respectively. The mapping of the class $c_{t1}$ is provided by $\mu_1$, while the mapping of the class $c_{t2}$ is provided by $\mu_2$.

\[
\begin{align*}
& t_1 = (c_{t1}, \text{rdfs:subClassOf}, ?x) \\
& t_2 = (c_{t2}, \text{rdfs:subClassOf}, ?x) \\
& \mu_1 : c_{t1} \equiv \text{trg:ComputerScience} \\
& \mu_2 : c_{t2} \equiv \text{trg:Mathematics}
\end{align*}
\]

Thus,

\[
S^s_c(t, \mu) = S^s_c(t_1, \mu_1) \text{ AND } S^s_c(t_2, \mu_2)
\]

The mappings $\mu_1$ and $\mu_2$ are of type $c_s \rightarrow c_t$. Thus, using the parameters defined above, as well as the function (7.9) for the rewriting of the triple patterns $t_1$ and $t_2$, the initial triple pattern $t$ is rewritten as follows:

\[
S^s_c(t, \mu) = S^s_c(t_1, \mu_1) \text{ AND } S^s_c(t_2, \mu_2)
\]

\[
= (\text{trg:ComputerScience, rdfs:subClassOf, ?x}) \text{ AND } (\text{trg:Mathematics, rdfs:subClassOf, ?x})
\]

**Example 7.4.** Consider the query posed over the source ontology of Figure 4.1: “Return the superclasses of the class `NewPublication`”. The SPARQL syntax of the source query is shown below:

```sparql
@PREFIX src: <http://www.owl-ontologies.com/SourceOntology.owl#>.
@PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>.

SELECT ?x
WHERE {src:NewPublication rdfs:subClassOf ?x.}
```

In order to rewrite the SPARQL query for posing it over the target ontology of Figure 4.1, we have to rewrite the triple pattern $t = (\text{src:NewPublication, rdfs:subClassOf, ?x})$ by its subject, predicate and object parts. Taking into consideration a mapping $\mu$ of the triple pattern’s subject part, the result of the triple pattern’s $t$ rewriting by its subject part is provided by invoking the function (7.9).
7.1. REWRITING BASED ON TRIPLE PATTERN'S SUBJECT PART

\[ t = (\text{src} : \text{NewPublication}, \text{rdfs} : \text{subClassOf}, ?x) \]

\[ \mu : \text{src} : \text{NewPublication} \equiv \text{trg} : \text{Computing} \sqcap \text{trg} : \text{NewRelease} \]

The mapping \( \mu \) is of type \( c_s \rightarrow c_t \sqcap c_t \). Following the definition of the function (7.9), two triple patterns \( t_1 \) and \( t_2 \) are created and the complex mapping \( \mu \) is decomposed into the mappings \( \mu_1 \) and \( \mu_2 \). The triple patterns \( t_1 \) and \( t_2 \) contain the classes \( c_{t_1} \) and \( c_{t_2} \) on their subject part, respectively. The mapping of the class \( c_{t_1} \) is provided by \( \mu_1 \), while the mapping of the class \( c_{t_2} \) is provided by \( \mu_2 \).

\[ t_1 = (c_{t_1}, \text{rdfs} : \text{subClassOf}, ?x) \]
\[ t_2 = (c_{t_2}, \text{rdfs} : \text{subClassOf}, ?x) \]
\[ \mu_1 : c_{t_1} \equiv \text{trg} : \text{Computing} \]
\[ \mu_2 : c_{t_2} \equiv \text{trg} : \text{NewRelease} \]

Thus,

\[ S^\circ_c(t, \mu) = S^\circ_c(t_1, \mu_1) \cup S^\circ_c(t_2, \mu_2) \]

The mappings \( \mu_1 \) and \( \mu_2 \) are of type \( c_s \rightarrow c_t \). Thus, using the parameters defined above, as well as the function (7.9) for the rewriting of the triple patterns \( t_1 \) and \( t_2 \), the initial triple pattern \( t \) is rewritten as follows:

\[ S^\circ_c(t, \mu) = S^\circ_c(t_1, \mu_1) \cup S^\circ_c(t_2, \mu_2) \]
\[ = (\text{trg} : \text{Computing}, \text{rdfs} : \text{subClassOf}, ?x) \cup (\text{trg} : \text{NewRelease}, \text{rdfs} : \text{subClassOf}, ?x) \]

**Example 7.5.** Consider the query posed over the source ontology of Figure 4.1: “Return the classes which are specified to be equivalent to the class Science”. The SPARQL syntax of the source query is shown below:

```sparql
@PREFIX src: <http://www.ontologies.com/SourceOntology.owl#>.
@PREFIX owl: <http://www.w3.org/2002/07/owl#>.

SELECT ?x
WHERE {src:Science owl:equivalentClass ?x.}
```
In order to rewrite the SPARQL query for posing it over the target ontology of Figure 4.1, we have to rewrite the triple pattern \( t = (\text{src} : \text{Science}, \text{owl} : \text{equivalentClass}, ?x) \) by its subject, predicate and object parts. Taking into consideration a mapping \( \mu \) of the triple pattern’s subject part, the result of the triple pattern’s rewriting by its subject part is provided by invoking the function (7.9).

\[
\mu : \text{src} : \text{Science} \equiv \text{trg} : \text{ComputerScience} \sqcup \text{trg} : \text{Mathematics}
\]

The mapping \( \mu \) is of type \( c_s \rightarrow c_{t1} \sqcup c_{t2} \). Following the definition of the function (7.9), two triple patterns \( t_1 \) and \( t_2 \) are created and the complex mapping \( \mu \) is decomposed into the mappings \( \mu_1 \) and \( \mu_2 \). The triple patterns \( t_1 \) and \( t_2 \) contain the classes \( c_{t1} \) and \( c_{t2} \) on their subject part, respectively. The mapping of the class \( c_{t1} \) is provided by \( \mu_1 \), while the mapping of the class \( c_{t2} \) is provided by \( \mu_2 \).

\[
t_1 = (c_{t1}, \text{owl} : \text{equivalentClass}, ?x)
\]
\[
t_2 = (c_{t2}, \text{owl} : \text{equivalentClass}, ?x)
\]
\[
\mu_1 : c_{t1} \equiv \text{trg} : \text{ComputerScience}
\]
\[
\mu_2 : c_{t2} \equiv \text{trg} : \text{Mathematics}
\]

Thus,

\[
S_c^\ast(t, \mu) = S_c^\ast(t_1, \mu_1) \text{ AND } S_c^\ast(t_2, \mu_2)
\]

The mappings \( \mu_1 \) and \( \mu_2 \) are of type \( c_s \rightarrow c_t \). Thus, using the parameters defined above, as well as the function (7.9) for the rewriting of the triple patterns \( t_1 \) and \( t_2 \), the initial triple pattern \( t \) is rewritten as follows:

\[
S_c^\ast(t, \mu) = S_c^\ast(t_1, \mu_1) \text{ AND } S_c^\ast(t_2, \mu_2)
\]

\[
= (\text{trg} : \text{ComputerScience}, \text{owl} : \text{equivalentClass}, ?x) \text{ AND } (\text{trg} : \text{Mathematics}, \text{owl} : \text{equivalentClass}, ?x)
\]
Rewriting based on object property mapping. Let $op_s$ be an object property from the source ontology which is mapped to an object property expression from the target ontology. Having a Schema Triple Pattern $t = (op_s, predicate, object)$ with $op_s$ in its subject part, an RDF/RDFS/OWL property in its predicate and anything in its object part, we can rewrite it by its subject part, using a predefined mapping $\mu$ and the function (7.10).

\[
S_{op_s}(t, \mu) = \begin{cases} 
(op_s, predicate, object) & \text{if } \mu : op_s \rightarrow op_i \\
S_{op_s}(t_1, \mu_1) \cup S_{op_s}(t_2, \mu_2) & \text{if } \mu : op_s \rightarrow op_1 \cap op_2 \text{ and } \\
& \text{predicate} = \text{rdf} : \text{subPropertyOf}, \\
& \text{where } t_1 = (op_1, predicate, object), \\
& \mu_1 : op_1 \equiv OPE_{i_1}, \\
& \text{and } t_2 = (op_2, predicate, object), \\
& \mu_2 : op_2 \equiv OPE_{i_2} \ \\
S_{op_s}(t_1, \mu_1) \cap S_{op_s}(t_2, \mu_2) & \text{if } \mu : op_s \rightarrow [\cap \cup] op_1 \cap op_2 \text{ and } \\
& \text{predicate} \in SSP^p, \\
& \text{or if } \mu : op_s \rightarrow op_1 \cup op_2 \text{ and } \\
& \text{predicate} = \text{rdf} : \text{subPropertyOf}, \\
& \text{where } t_1 = (op_1, predicate, object), \\
& \mu_1 : op_1 \equiv OPE_{i_1}, \\
& \text{and } t_2 = (op_2, predicate, object), \\
& \mu_2 : op_2 \equiv OPE_{i_2} 
\end{cases}
\]

Rewriting based on datatype property mapping. Let $dp_s$ be a datatype property from the source ontology which is mapped to a datatype property expression from the target ontology. Having a Schema Triple Pattern $t = (dp_s, predicate, object)$ with $dp_s$ in its subject part, an RDF/RDFS/OWL property in its predicate and anything in its object part, we can rewrite it by its subject part, using a predefined mapping $\mu$ and the function (7.11).
(dp, predicate, object) if \( \mu : dp_s \rightarrow dp_t \)

\[
S_{dp}(t_1, \mu_1) \text{ UNION } S_{dp}(t_2, \mu_2)
\]

if \( \mu : dp_s \rightarrow dp_{t_1} \cap dp_{t_2} \) and

\( \text{predicate} = \text{rdfs : subPropertyOf} \),

where \( t_1 = (dp_{t_1}, \text{predicate}, \text{object}) \),

\( \mu_1 : dp_{t_1} \equiv DPE_{t_1} \),

and \( t_2 = (dp_{t_2}, \text{predicate}, \text{object}) \),

\( \mu_2 : dp_{t_2} \equiv DPE_{t_2} \)

\( S_{dp}(t, \mu) = \) \( \{ \}

\[
S_{dp}(t_1, \mu_1) \text{ AND } S_{dp}(t_2, \mu_2)
\]

if \( \mu : dp_s \rightarrow dp_{t_1} \cap \cup dp_{t_2} \) and

\( \text{predicate} \in SSP' \),

or if \( \mu : dp_s \rightarrow dp_{t_1} \cup dp_{t_2} \) and

\( \text{predicate} = \text{rdfs : subPropertyOf} \),

where \( t_1 = (dp_{t_1}, \text{predicate}, \text{object}) \),

\( \mu_1 : dp_{t_1} \equiv DPE_{t_1} \),

and \( t_2 = (dp_{t_2}, \text{predicate}, \text{object}) \),

\( \mu_2 : dp_{t_2} \equiv DPE_{t_2} \)

(7.11)

The functions (7.10) and (7.11) are used similarly with the function (7.9), which performs triple pattern rewriting by subject part, based on a class mapping.

**Rewriting based on individual mapping.** Let \( i_s \) be an individual from the source ontology which is mapped to an individual \( i_t \) from the target ontology. Having a Schema Triple Pattern \( t = (i_s, \text{predicate}, \text{object}) \) with \( i_s \) in its subject part, an RDF/RDFS/OWL property in its predicate and anything in its object part, we can rewrite it by its subject part, using a predefined mapping \( \mu \) and the function (7.12).

\[
S_{i}(t, \mu) = (i_t, \text{predicate}, \text{object}) \text{ if } \mu : i_s \equiv i_t
\]

(7.12)

In Lemma 7.1 we summarize the functions presented in this section, which are used for the rewriting of a Schema Triple Pattern based on a mapping for the triple pattern’s subject part.

**Lemma 7.1.** Let \( i_s \) be an individual, \( c_s \) be a class, \( op_s \) be an object property and \( dp_s \) be a datatype property from the source ontology. Having a Schema Triple Pattern \( t \) and a predefined mapping \( \mu \) for its subject part, we can rewrite it by its subject, by invoking the function (7.13).
7.2 Rewriting based on triple pattern’s object part

Unlike the Data Triple Patterns, a property can appear in the object part of a Schema Triple Pattern.

**Rewriting based on class mapping.** Let $c_s$ be a class from the source ontology which is mapped to a class expression from the target ontology. Having a Schema Triple Pattern $t = (\text{subject}, \text{predicate}, c_s)$ with $c_s$ in its object part, an RDF/RDFS/OWL property in its predicate and anything in its subject part, we can rewrite it by its object part, using a predefined mapping $\mu$ and the function (7.14).

$$
S^o_c(t, \mu) = \begin{cases} 
S^i_s(t, \mu) & \text{if } t = (i_s, \text{predicate}, \text{object}) \\
S^i_c(t, \mu) & \text{if } t = (c_s, \text{predicate}, \text{object}) \\
S^i_{op}(t, \mu) & \text{if } t = (op_s, \text{predicate}, \text{object}) \\
S^i_{dp}(t, \mu) & \text{if } t = (dp_s, \text{predicate}, \text{object})
\end{cases}
$$

(7.13)

$$
S^o_c(t, \mu) = \begin{cases} 
\{ (\text{subject}, \text{predicate}, c_1) \} & \text{if } \mu : c_s \rightarrow c_1 \\
S^o_c(t_1, \mu_1) \text{ UNION } S^o_c(t_2, \mu_2) & \text{if } \mu : c_s \rightarrow c_{11} \cup c_{12} \text{ and } \text{predicate} = \text{rdf} : \text{subClassOf}, \\
& \text{where } t_1 = (\text{subject}, \text{predicate}, c_{11}), \\
& \mu_1 : c_{11} \equiv CE_{t_1}, \\
& \text{and } t_2 = (\text{subject}, \text{predicate}, c_{12}), \\
& \mu_2 : c_{12} \equiv CE_{t_2} \\
\{ (\text{subject}, \text{predicate}, c_1) \} \text{ AND } S^o_c(t_2, \mu_2) & \text{if } \mu : c_s \rightarrow c_{11} [\cap | \cup] c_{12} \text{ and } \text{predicate} \in SSP_c, \\
& \text{or if } \mu : c_s \rightarrow c_{11} \cap c_{12} \text{ and } \text{predicate} = \text{rdf} : \text{subClassOf}, \\
& \text{where } t_1 = (\text{subject}, \text{predicate}, c_{11}), \\
& \mu_1 : c_{11} \equiv CE_{t_1}, \\
& \text{and } t_2 = (\text{subject}, \text{predicate}, c_{12}), \\
& \mu_2 : c_{12} \equiv CE_{t_2}
\end{cases}
$$

(7.14)
Example 7.6. Consider the query posed over the source ontology of Figure 4.1: “Return the subclasses of the class Science”. The SPARQL syntax of the source query is shown below:

```sparql
@PREFIX src: <http://www.owl-ontologies.com/SourceOntology.owl#>.
@PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
SELECT ?x
WHERE {?x rdfs:subClassOf src:Science.}
```

In order to rewrite the SPARQL query for posing it over the target ontology of Figure 4.1, we have to rewrite the triple pattern $t = (\text{?}x; \text{rdfs:subClassOf}; \text{src:Science})$ by its subject, predicate and object parts. Taking into consideration a mapping $\mu$ of the triple pattern’s object part, the result of the triple pattern’s $t$ rewriting by its object part is provided by invoking the function (7.14).

$$t = (\text{?}x, \text{rdfs:subClassOf}, \text{src:Science})$$

$$\mu : \text{src:Science} \equiv \text{trg:ComputerScience} \cup \text{trg:Mathematics}$$

The mapping $\mu$ is of type $c_s \rightarrow c_t^1 \sqcup c_t^2$. Following the definition of the function (7.14), two triple patterns $t_1$ and $t_2$ are created and the complex mapping $\mu$ is decomposed into the mappings $\mu_1$ and $\mu_2$. The triple patterns $t_1$ and $t_2$ contain the classes $c_t^1$ and $c_t^2$ on their object part, respectively. The mapping of the class $c_t^1$ is provided by $\mu_1$, while the mapping of the class $c_t^2$ is provided by $\mu_2$.

$$t_1 = (\text{?}x, \text{rdfs:subClassOf}, c_t^1)$$

$$t_2 = (\text{?}x, \text{rdfs:subClassOf}, c_t^2)$$

$$\mu_1 : c_t^1 \equiv \text{trg:ComputerScience}$$

$$\mu_2 : c_t^2 \equiv \text{trg:Mathematics}$$

Thus,

$$S_o^c(t, \mu) = S_o^c(t_1, \mu_1) \cup S_o^c(t_2, \mu_2)$$

The mappings $\mu_1$ and $\mu_2$ are of type $c_s \rightarrow c_t$. Thus, using the parameters defined above, as well as the function (7.14) for the rewriting of the triple patterns $t_1$ and $t_2$, the initial triple pattern $t$ is rewritten as follows:

$$S_o^c(t, \mu) = S_o^c(t_1, \mu_1) \cup S_o^c(t_2, \mu_2)$$

$$= (\text{?}x, \text{rdfs:subClassOf}, \text{trg:ComputerScience}) \cup (\text{?}x, \text{rdfs:subClassOf}, \text{trg:Mathematics})$$
Example 7.7. Consider the query posed over the source ontology of Figure 4.1: “Return the subclasses of the class NewPublication”. The SPARQL syntax of the source query is shown below:

```
@PREFIX src: <http://www.owl-ontologies.com/SourceOntology.owl#>.
@PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>.

SELECT ?x
WHERE {?x rdfs:subClassOf src:NewPublication.}
```

In order to rewrite the SPARQL query for posing it over the target ontology of Figure 4.1, we have to rewrite the triple pattern \( t = (\text{?}_x; \text{rdfs:subClassOf}; \text{src:NewPublication}) \) by its subject, predicate and object parts. Taking into consideration a mapping \( \mu \) of the triple pattern’s object part, the result of the triple pattern’s \( t \) rewriting by its object part is provided by invoking the function (7.14).

\[
t = (\text{?}_x, \text{rdfs:subClassOf}, \text{src:NewPublication})
\]

\[
\mu : \text{src:NewPublication} \equiv \text{trg:Computing} \cap \text{trg:NewRelease}
\]

The mapping \( \mu \) is of type \( c_s \rightarrow c_t \cap c_{t_2} \). Following the definition of the function (7.14), two triple patterns \( t_1 \) and \( t_2 \) are created and the complex mapping \( \mu \) is decomposed into the mappings \( \mu_1 \) and \( \mu_2 \). The triple patterns \( t_1 \) and \( t_2 \) contain the classes \( c_{t_1} \) and \( c_{t_2} \) on their object part, respectively. The mapping of the class \( c_{t_1} \) is provided by \( \mu_1 \), while the mapping of the class \( c_{t_2} \) is provided by \( \mu_2 \).

\[
t_1 = (\text{?}_x, \text{rdfs:subClassOf}, c_{t_1})
\]

\[
t_2 = (\text{?}_x, \text{rdfs:subClassOf}, c_{t_2})
\]

\[
\mu_1 : c_{t_1} \equiv \text{trg:Computing}
\]

\[
\mu_2 : c_{t_2} \equiv \text{trg:NewRelease}
\]

Thus,

\[
S^o_c(t, \mu) = S^o_c(t_1, \mu_1) \land S^o_c(t_2, \mu_2)
\]

The mappings \( \mu_1 \) and \( \mu_2 \) are of type \( c_s \rightarrow c_t \). Thus, using the parameters defined above, as well as the function (7.14) for the rewriting of the triple patterns \( t_1 \) and \( t_2 \), the initial triple pattern \( t \) is rewritten as follows:

\[
S^o_c(t, \mu) = S^o_c(t_1, \mu_1) \land S^o_c(t_2, \mu_2)
\]

\[
= (\text{?}_x, \text{rdfs:subClassOf}, \text{trg:Computing}) \land (\text{?}_x, \text{rdfs:subClassOf}, \text{trg:NewRelease})
\]
Example 7.8. Consider the query posed over the source ontology of Figure 4.1: “Return the classes which have been specified to be disjoint with the class Science”. The SPARQL syntax of the source query is shown below:

@PREFIX src: <http://www.owl-ontologies.com/SourceOntology.owl#>.
@PREFIX owl: <http://www.w3.org/2002/07/owl#>.

SELECT ?x
WHERE {?x owl:disjointWith src:Science.}

In order to rewrite the SPARQL query for posing it over the target ontology of Figure 4.1, we have to rewrite the triple pattern \( t = (?x, owl:disjointWith, src:Science) \) by its subject, predicate and object parts. Taking into consideration a mapping \( \mu \) of the triple pattern’s object part, the result of the triple pattern’s \( t \) rewriting by its object part is provided by invoking the function (7.14).

\[
t = (?x, owl:disjointWith, src:Science)
\]

\[
\mu : src:Science \equiv trg:ComputerScience \cup trg:Mathematics
\]

The mapping \( \mu \) is of type \( c_s \rightarrow c_t \). Following the definition of the function (7.14), two triple patterns \( t_1 \) and \( t_2 \) are created and the complex mapping \( \mu \) is decomposed into the mappings \( \mu_1 \) and \( \mu_2 \). The triple patterns \( t_1 \) and \( t_2 \) contain the classes \( c_1 \) and \( c_2 \) on their object part, respectively. The mapping of the class \( c_1 \) is provided by \( \mu_1 \), while the mapping of the class \( c_2 \) is provided by \( \mu_2 \).

\[
t_1 = (?x, owl:disjointWith, c_1)
\]

\[
t_2 = (?x, owl:disjointWith, c_2)
\]

\[
\mu_1 : c_1 \equiv trg:ComputerScience
\]

\[
\mu_2 : c_2 \equiv trg:Mathematics
\]

Thus,

\[
S.o_c(t, \mu) = S.o_c(t_1, \mu_1) \text{ AND } S.o_c(t_2, \mu_2)
\]

The mappings \( \mu_1 \) and \( \mu_2 \) are of type \( c_s \rightarrow c_t \). Thus, using the parameters defined above, as well as the function (7.14) for the rewriting of the triple patterns \( t_1 \) and \( t_2 \), the initial triple pattern \( t \) is rewritten as follows:

\[
S.o_c(t, \mu) = S.o_c(t_1, \mu_1) \text{ AND } S.o_c(t_2, \mu_2)
\]

\[
= (?x, owl:disjointWith, trg:ComputerScience) \text{ AND } (?x, owl:disjointWith, trg:Mathematics)
\]
Rewriting based on object property mapping. Let \( op_s \) be an object property from the source ontology which is mapped to an object property expression from the target ontology. Having a Schema Triple Pattern \( t = (subject, predicate, op_s) \) with \( op_s \) in its object part, an RDF/RDFS/OWL property in its predicate and anything in its subject part, we can rewrite it by its object part, using a predefined mapping \( \mu \) and the function (7.15).

\[
S_{op}(t, \mu) = \begin{cases} 
(subject, predicate, op_s) & \text{if } \mu : op_s \rightarrow op_t \\
S_{op}(t_1, \mu_1) \cup S_{op}(t_2, \mu_2) & \text{if } \mu : op_s \rightarrow op_{t_1} \sqcup op_{t_2} \text{ and } \text{predicate} = \text{rdfs:subPropertyOf}, \\
\text{where } t_1 = (subject, predicate, op_{t_1}), \\
\mu_1 : op_{t_1} \equiv OPE_{t_1}, \\
\text{and } t_2 = (subject, predicate, op_{t_2}), \\
\mu_2 : op_{t_2} \equiv OPE_{t_2} \\
S_{op}(t_1, \mu_1) \wedge S_{op}(t_2, \mu_2) & \text{if } \mu : op_s \rightarrow op_{t_1} \cap \sqcup op_{t_2} \text{ and } \text{predicate} \in SSP'_{p}, \\
\text{or if } \mu : op_s \rightarrow op_{t_1} \sqsubseteq op_{t_2} \text{ and } \text{predicate} = \text{rdfs:subPropertyOf}, \\
\text{where } t_1 = (subject, predicate, op_{t_1}), \\
\mu_1 : op_{t_1} \equiv OPE_{t_1}, \\
\text{and } t_2 = (subject, predicate, op_{t_2}), \\
\mu_2 : op_{t_2} \equiv OPE_{t_2}
\end{cases}
\]  

(7.15)

Rewriting based on datatype property mapping. Let \( dp_s \) be a datatype property from the source ontology which is mapped to a datatype property expression from the target ontology. Having a Schema Triple Pattern \( t = (subject, predicate, dp_s) \) with \( dp_s \) in its object part, an RDF/RDFS/OWL property in its predicate and anything in its subject part, we can rewrite it by its object part, using a predefined mapping \( \mu \) and the function (7.16).

\[
\]
The functions (7.15) and (7.16) are used similarly with the function (7.14), which performs triple pattern rewriting by object part, based on a class mapping.

**Rewriting based on individual mapping.** Let $i_s$ be an individual from the source ontology which is mapped to an individual $i_t$ from the target ontology. Having a Schema Triple Pattern $t = (\text{subject}, \text{predicate}, i_s)$ with $i_s$ in its object part, an RDF/RDFS/OWL property in its predicate and anything in its subject part, we can rewrite it by its object part, using a predefined mapping $\mu$ and the function (7.17).

\[
S_{dp}(t, \mu) = \begin{cases} 
  (\text{subject}, \text{predicate}, dp_t) & \text{if } \mu : dp_s \rightarrow dp_t \\
  S_{dp}^a(t_1, \mu_1) \cup S_{dp}^a(t_2, \mu_2) & \text{if } \mu : dp_s \rightarrow dp_{t_1} \cup dp_{t_2} \text{ and } \\
  \text{predicate} = \text{rdf}s : \text{subPropertyOf}, \\
  \text{where } t_1 = (\text{subject}, \text{predicate}, dp_{t_1}), \\
  \mu_1 : dp_{t_1} \equiv \text{DPE}_{t_1}, \\
  \text{and } t_2 = (\text{subject}, \text{predicate}, dp_{t_2}), \\
  \mu_2 : dp_{t_2} \equiv \text{DPE}_{t_2} \\
  S_{dp}^a(t_1, \mu_1) \text{ AND } S_{dp}^a(t_2, \mu_2) & \text{if } \mu : dp_s \rightarrow dp_{t_1} \cap dp_{t_2} \text{ and } \\
  \text{predicate} \in \text{SSP}'', \\
  \text{or if } \mu : dp_s \rightarrow dp_{t_1} \cap dp_{t_2} \text{ and } \\
  \text{predicate} = \text{rdf}s : \text{subPropertyOf}, \\
  \text{where } t_1 = (\text{subject}, \text{predicate}, dp_{t_1}), \\
  \mu_1 : dp_{t_1} \equiv \text{DPE}_{t_1}, \\
  \text{and } t_2 = (\text{subject}, \text{predicate}, dp_{t_2}), \\
  \mu_2 : dp_{t_2} \equiv \text{DPE}_{t_2}
\end{cases}
\]  

(7.16)

In Lemma 7.2 we summarize the functions presented in this section, which are used for the rewriting of a Schema Triple Pattern based on a mapping for the triple pattern’s object part.

**Lemma 7.2.** Let $i_s$ be an individual, $c_s$ be a class, $op_s$ be an object property and $dp_s$ be a datatype property from the source ontology. Having a Schema Triple Pattern $t$ and a predefined mapping $\mu$ for its object part, we can rewrite it by its object, by invoking the function (7.18).
7.2. REWRITING BASED ON TRIPLE PATTERN’S OBJECT PART

\[
S_i'(t, \mu) = \begin{cases} 
S_i'(t, \mu) & \text{if } t = (\text{subject}, \text{predicate}, i, s) \\
S_c'(t, \mu) & \text{if } t = (\text{subject}, \text{predicate}, c, s) \\
S_{op}'(t, \mu) & \text{if } t = (\text{subject}, \text{predicate}, op, s) \\
S_{dp}'(t, \mu) & \text{if } t = (\text{subject}, \text{predicate}, dp, s) 
\end{cases} 
\tag{7.18}
\]
In this chapter, we present the algorithms performing graph pattern rewriting, based on a set of predefined mappings. Algorithm 1 takes as input a SPARQL query’s graph pattern $GP_{in}$, as well as a set of mappings $M$. Moreover, it uses Algorithm 2 in order to perform triple pattern rewriting by exploiting mappings for a specific triple pattern part (i.e. subject, predicate, or object), according to a specified parameter.

In the first step, the algorithm rewrites every Filter expression inside the graph pattern (line 2). The SPARQL variables, literal constants, operators ($\&\&$, $|$, $!$, $=\neq$, $>$, $<$, $\geq$, $\leq$, $\pm$, $\ast$, $=$) and built-in functions (e.g. $\text{bound}$, $\text{isIRI}$, $\text{isLiteral}$, $\text{datatype}$, $\text{lang}$, $\text{str}$, $\text{regex}$) which may appear inside a Filter expression remain the same during the rewriting process. For class IRIs and property IRIs which may appear inside a Filter expression of a SPARQL query, we use 1:1 cardinality mappings for the expression rewriting. This poses a minor limitation, considering that an IRI can appear inside a Filter expression only for equality and inequality operations. Thus, the rewriting of a Filter expression is performed by substituting any IRIs that refer to a class, property, or individual, according to the specified mappings.

Then, the algorithm rewrites the graph pattern triple pattern by triple pattern, using the mappings of the triple patterns’ predicate parts (line 3). Similarly, it rewrites the resulted graph pattern, using the mappings of the triple patterns’ object parts (line 4) and then using the mappings of the triple patterns’ subject parts (line 5). Finally, after removing any unnecessary brackets (line 6) the resulted graph pattern (line 7) is ready to replace the graph pattern of the initial query which has been posed over the source ontology, in order for the resulted query to be posed over the target ontology.

Algorithm 2 rewrites the triple patterns of a graph pattern, using mappings for a specific triple pattern part, as well as the Data and Schema Triple Pattern rewriting functions presented in Chapter 6 and Chapter 7, respectively. It takes as input a SPARQL graph pattern $GP_{in}$, a set of mappings $M$, as well as the triple pattern part $x$ which will be used for the rewriting. The initial graph pattern’s operators (AND, UNION, OPTIONAL,
Algorithm 1 Graph Pattern Rewriting (\(GP_{in}\): input graph pattern, \(M\): mapping set)

1: let \(GP_{out}\) be the rewriting result of \(GP_{in}\)
2: \(GP_{out} \leftarrow GP_{in}\) after replacing any user defined IRIs (class, property, individual) inside \(\text{FILTER}\) expressions using the 1:1 cardinality mappings of \(M\)
3: \(GP_{out} \leftarrow \text{Triple Pattern Rewriting} (GP_{out}, M, \text{predicate})\)
4: \(GP_{out} \leftarrow \text{Triple Pattern Rewriting} (GP_{out}, M, \text{object})\)
5: \(GP_{out} \leftarrow \text{Triple Pattern Rewriting} (GP_{out}, M, \text{subject})\)
6: \(GP_{out} \leftarrow GP_{out}\) after removing any unnecessary brackets
7: return \(GP_{out}\)

\(\text{FILTER}\) remain the same during the rewriting process, while SPARQL variables, blank nodes, literal constants and RDF/RDFS/OWL IRIs which may appear in a triple pattern part do not affect the rewriting procedure. This means that the SPARQL variables of the initial query appear also in the rewritten query.

Example 8.1. Consider the query posed over the source ontology of Figure 4.1: “Return the ids of the products named Linux”. The SPARQL syntax of the source query is shown below:

\[
\begin{align*}
\text{@PREFIX src: } & <\text{http://www.ontologies.com/SourceOntology.owl#}>. \\
\text{@PREFIX rdf: } & <\text{http://www.w3.org/1999/02/22-rdf-syntax-ns#}>. \\
\text{SELECT } & \text{?id} \\
\text{WHERE } & \{ ?x \text{ rdf:type src:Product}. \\
& \quad ?x \text{ src:id } \text{?id}. \\
& \quad ?x \text{ src:name } \text{?name}. \\
& \quad \text{FILTER(?name=“Linux“)} \}
\end{align*}
\]

In order to rewrite the SPARQL query’s graph pattern \(GP_{in}\), Algorithm 1, as well as a set of predefined mappings \(M\) should be used. Let the available predefined mappings be the mappings \(\mu_1\), \(\mu_2\) and \(\mu_3\) presented below. The inputs of Algorithm 1 are \(GP_{in}\) and \(M\).

\[
GP_{in} = (?x, \text{rdf:type }, \text{src:Product}) \text{ AND } (?x, \text{src:id }, \text{?id}) \text{ AND } \\
(?x, \text{src:name }, \text{?name}) \text{ FILTER(?name = “Linux“)}
\]

\[
M = \{ \mu_1 : \text{src:Product } \sqsubseteq \text{trg:Textbook}, \\
\mu_2 : \text{src:id } \sqsubseteq \text{trg:isbn}, \\
\mu_3 : \text{src:name } \sqsubseteq \text{trg:title} \}
\]

\(GP_{in}\) contains a \(\text{FILTER}\) operation which does not affect the rewriting procedure since the filter’s expression consists of SPARQL variables and literal constants. Consequently,
Algorithm 2 Triple Pattern Rewriting \(GP_{in}:\) input graph pattern, \(M:\) mapping set, \(x:\) triple pattern part

Require: \(x \in \{\text{subject, predicate, object}\}\)

1. let \(x(t)\) be the \(x\) part of a triple pattern \(t\)
2. let \(\text{relax}(\mu)\) be the relaxed form of mapping \(\mu\)
3. let \(GP_{out}\) be the rewriting result of \(GP_{in}\)
4. for each basic graph pattern \(BGP\) in \(GP_{in}\) do
5. let \(GP_{temp1}\) be the rewriting result of \(BGP\)
6. for each triple pattern \(t\) in \(BGP\) do
7. let \(GP_{temp2}\) be the rewriting result of \(t\)
8. if \(x(t)\) is a variable, a blank node, a literal constant, or an RDF/RDFS/OWL property then
9. \(GP_{temp2} \leftarrow t\)
10. else
11. if \(t \in DTP\) then \{in case that \(t\) is a Data Triple Pattern\}
12. if \(x = \text{subject}\) then
13. let \(\mu_s \in M\) be the mapping of \(t\)'s subject
14. \(GP_{temp2} \leftarrow D^s_\mu(t, \mu_s)\)
15. else if \(x = \text{predicate}\) then
16. let \(\mu_p \in M\) be the mapping of \(t\)'s predicate
17. \(GP_{temp2} \leftarrow D^p_\mu(t, \mu_p)\)
18. else
19. let \(\mu_o \in M\) be the mapping of \(t\)'s object
20. \(GP_{temp2} \leftarrow D^o_\mu(t, \mu_o)\)
21. end if
22. else \{in case that \(t\) is a Schema Triple Pattern\}
23. if \(x = \text{subject}\) then
24. let \(\mu_s \in M\) be the mapping of \(t\)'s subject
25. \(\mu'_s \leftarrow \text{relax}(\mu_s)\)
26. \(GP_{temp2} \leftarrow S^s_\mu(t, \mu'_s)\)
27. else
28. if \(x = \text{object}\) then
29. let \(\mu_o \in M\) be the mapping of \(t\)'s object
30. \(\mu'_o \leftarrow \text{relax}(\mu_o)\)
31. \(GP_{temp2} \leftarrow S^o_\mu(t, \mu_o)\)
32. end if
33. end if
34. end if
35. end if
36. \(GP_{temp1} \leftarrow GP_{temp1}\) after appending \(GP_{temp2}\)
37. end for
38. \(GP_{temp1} \leftarrow GP_{temp1}\) after applying any filters according to the \(BGP\) form
39. \(GP_{temp1} \leftarrow \{GP_{temp1}\}\) after applying any brackets in order to form the graph pattern precedence according to the \(GP_{in}\) form
40. \(GP_{out} \leftarrow GP_{out}\) after appending \(GP_{temp1}\)
41. \(GP_{out} \leftarrow GP_{out}\) after appending any operators/filters to the rewritten graph pattern according to the \(GP_{in}\) form
42. end for
43. return \(GP_{out}\)
the algorithm proceeds to the invocation of Algorithm 2 (step 3) in order to rewrite each triple pattern of $GP_m$, using the mappings of the triple patterns’ predicate parts. The input of Algorithm 2 is the initial graph pattern $GP_m$, as well as the set of mappings $M$.

The graph pattern $GP_m$, is actually a basic graph pattern since it consists of a triple pattern sequence followed by a FILTER operation. Algorithm 2 rewrites the basic graph pattern $GP_m$ triple pattern by triple pattern, using the mappings of the triple patterns’ predicate parts. The triple pattern $t_1 = (?x, rdf : type, src : Product)$ remains the same after the rewriting process, since its predicate part consists of the RDF property rdf : type. On the contrary, the rewriting result of the triple pattern $t_2 = (?x, src : id, ?id)$, as well as the rewriting result of the triple pattern $t_3 = (?x, src : name, ?name)$ are provided by invoking the function (6.11).

\[
D^p_d(t_2, \mu_2) = D^p_d((?x, src : id, ?id), \mu_2) = (?x, trg : isbn, ?id)
\]

\[
D^p_d(t_3, \mu_3) = D^p_d((?x, src : name, ?name), \mu_3) = (?x, trg : title, ?name)
\]

Consequently, the output of Algorithm 2 is a graph pattern, having its triple patterns rewritten by their predicate part and is presented below:

\[
GP_p = (?x, rdf : type, src : Product) \text{ AND } (?x, trg : isbn, ?id) \text{ AND } (?x, trg : title, ?name) \text{ FILTER(?name = "Linux")}
\]

Then, Algorithm 1 uses Algorithm 2 in order to rewrite the triple patterns of $GP_p$ by their object parts (step 4). All the triple patterns except of $t_1$ remain the same, since they contain a SPARQL variable on their object part. The rewriting result of the triple pattern $t_1$, is provided by invoking the function (6.8).

\[
D^o_c(t_1, \mu_1) = D^o_c((?x, rdf : type, src : Product), \mu_1) = (?x, rdf : type, trg : Textbook)
\]

Consequently, the output of second invocation of Algorithm 2 is a graph pattern, having its triple patterns rewritten by their object part and is presented below:

\[
GP_o = (?x, rdf : type, trg : Textbook) \text{ AND } (?x, trg : isbn, ?id) \text{ AND } (?x, trg : title, ?name) \text{ FILTER(?name = "Linux")}
\]
Finally, Algorithm 1 proceeds to step 5 (Algorithm 2 invocation) in order to rewrite the triple patterns of $GP_o$ by their subject parts. However, the resulted graph pattern $GP_s$ is the same with $GP_o$, since every triple pattern of $GP_o$ contains a SPARQL variable in its subject part.

$$GP_s = (?x, rdf : type, trg : Textbook) \land (?x, trg : isbn, ?id) \land \neg (?x, trg : title, ?name) \land \text{FILTER}(name = "Linux")$$

After removing any unnecessary brackets from $GP_s$ (step 6), the rewritten SPARQL query is provided by replacing the initial query’s graph pattern with $GP_s$. Consequently, the SPARQL query which will be posed over the target ontology of Figure 4.1, is presented below:

\begin{verbatim}
@PREFIX trg: <http://www.ontologies.com/TargetOntology.owl#>.
@PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
SELECT ?id
WHERE { ?x rdf:type trg:Textbook.
  ?x trg:isbn ?id.
  FILTER(?name="Linux")}
\end{verbatim}

**Example 8.2.** Consider the query posed over the source ontology of Figure 4.1: “Return the individuals of every class which is specified to be subclass of the class NewPublication”. The SPARQL syntax of the source query is shown below:

\begin{verbatim}
@PREFIX src: <http://www.ontologies.com/SourceOntology.owl#>.
@PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
SELECT ?x
WHERE { ?x rdf:type ?y.
  ?y rdfs:subClassOf src:NewPublication.}
\end{verbatim}

In order to rewrite the SPARQL query’s graph pattern $GP_{in}$, Algorithm 1, as well as a set of predefined mappings $\mathcal{M}$ should be used. Let the available predefined mappings be the mapping $\mu$ presented below. The inputs of Algorithm 1 are $GP_{in}$ and $\mathcal{M}$.

$$GP_{in} = (?x, rdf : type, ?y) \land (?x, rdfs : subClassOf, src : Science)$$
\[ M = \{ \mu : \text{src} : \text{NewPublication} \equiv \text{trg} : \text{Computing} \land \text{trg} : \text{NewRelease} \} \]

\( GP_{in} \) does not contain any \textsc{filter} operations, thus the algorithm proceeds to the invocation of Algorithm 2 (step 3) in order to rewrite each triple pattern of \( GP_{in} \), using the mappings of the triple patterns’ predicate parts. The input of Algorithm 2 is the initial graph pattern \( GP_{in} \), as well as the set of mappings \( M \).

The graph pattern \( GP_{in} \), is actually a basic graph pattern since it consists of a triple pattern sequence. Algorithm 2 rewrites the basic graph pattern \( GP_{in} \) triple pattern by triple pattern, using the mappings of the triple patterns’ predicate parts. However, all the triple patterns remain the same, since they contain an RDF/S property on their predicate part. Consequently, the output of Algorithm 2 is presented below:

\[
GP_p = (\forall x, \text{rdf} : \text{type}, ?y) \land (\forall y, \text{rdf} s : \text{subClassOf}, \text{src} : \text{NewPublication})
\]

Then, Algorithm 1 uses Algorithm 2 in order to rewrite the triple patterns of \( GP_p \) by their object parts (step 4). The rewriting result of the triple pattern \( t_2 = (\forall y, \text{rdf} s : \text{subClassOf}, \text{src} : \text{NewPublication}) \), is provided by invoking the function (7.18), while the triple pattern \( t_1 = (\forall x, \text{rdf} : \text{type}, ?y) \) remains the same, since it contains a SPARQL variable on its object part.

\[
S^o_c(t_2, \mu) = S^o_c((\forall y, \text{rdf} s : \text{subClassOf}, \text{src} : \text{NewPublication}), \mu)
\]

\[
= (\forall y, \text{rdf} s : \text{subClassOf}, \text{trg} : \text{Computing}) \land (\forall y, \text{rdf} s : \text{subClassOf}, \text{trg} : \text{NewRelease})
\]

Consequently, the output of the second invocation of Algorithm 2 is a graph pattern, having its triple patterns rewritten by their object part and is presented below:

\[
GP_o = (\forall x, \text{rdf} : \text{type}, ?y) \land (\forall y, \text{rdf} s : \text{subClassOf}, \text{trg} : \text{Computing}) \land (\forall y, \text{rdf} s : \text{subClassOf}, \text{trg} : \text{NewRelease})
\]

Finally, Algorithm 1 proceeds to step 5 (Algorithm 2 invocation) in order to rewrite the triple patterns of \( GP_o \) by their subject parts. However, the resulted graph pattern \( GP_s \) is the same with \( GP_o \), since every triple pattern of \( GP_o \) contains a SPARQL variable in its subject part.

\[
GP_s = (\forall x, \text{rdf} : \text{type}, ?y) \land (\forall y, \text{rdf} s : \text{subClassOf}, \text{trg} : \text{Computing}) \land (\forall y, \text{rdf} s : \text{subClassOf}, \text{trg} : \text{NewRelease})
\]
After removing any unnecessary brackets from \( GP \) (step 6), the rewritten SPARQL query is provided by replacing the initial query’s graph pattern with \( GP \). Consequently, the SPARQL query which will be posed over the target ontology of Figure 4.1, is presented below:

\[
\text{SELECT } ?x \\
\text{WHERE } \{ ?x \text{ rdf:type } ?y. \\
\hspace{1em} ?y \text{ rdfs:subClassOf trg:Computing.} \\
\hspace{1em} ?y \text{ rdfs:subClassOf trg:NewRelease.} \}
\]

**Example 8.3.** Consider the query posed over the source ontology of Figure 4.1: “Return the titles of the pocket-sized scientific books”. The SPARQL syntax of the source query is shown below:

\[
\text{SELECT } ?name \\
\text{WHERE } \{ ?x \text{ src:name } ?name. \\
\hspace{1em} ?x \text{ src:type src:Science.} \\
\hspace{1em} ?x \text{ src:type src:Pocket.} \}
\]

In order to rewrite the SPARQL query’s graph pattern \( GP_{in} \), Algorithm 1, as well as a set of predefined mappings \( M \) should be used. Let the available predefined mappings be the mappings \( \mu_1, \mu_2 \) and \( \mu_3 \) presented below. The inputs of Algorithm 1 are \( GP_{in} \) and \( M \).

\[
GP_{in} = (?x, \text{src:name,?name}) \text{ AND } (?x, \text{rdf:type,src:Science}) \\
\hspace{1em} \text{AND } (?x, \text{rdf:type,src:Pocket})
\]

\[
M = \begin{cases} 
\mu_1 : \text{src:name } \sqsubseteq \text{trg:title}, \\
\mu_2 : \text{src:Science } \equiv \text{trg:ComputerScience } \sqcup \text{trg:Mathematics}, \\
\mu_3 : \text{src:Pocket } \equiv \forall \text{trg:Textbook.(trg:size } \leq 14) 
\end{cases}
\]

\( GP_{in} \) does not contain any \text{FILTER} operations, thus the algorithm proceeds to the invocation of Algorithm 2 (step 3) in order to rewrite each triple pattern of \( GP_{in} \), using
the mappings of the triple patterns’ predicate parts. The input of Algorithm 2 is the initial graph pattern \( GP_m \), as well as the set of mappings \( M \).

The graph pattern \( GP_m \), is actually a basic graph pattern since it consists of a triple pattern sequence. Algorithm 2 rewrites the basic graph pattern \( GP_m \) triple pattern by triple pattern, using the mappings of the triple patterns’ predicate parts. The triple patterns \( t_2 = (\text{x} : \text{type} : \text{Science}) \) and \( t_3 = (\text{x} : \text{type} : \text{Pocket}) \) remain the same since their predicate parts consist of the RDF property \( \text{rdf} : \text{type} \). On the contrary, the rewriting result of the triple pattern \( t_1 = (\text{x} : \text{name} : ?\text{name}) \), is provided by invoking the function (6.11).

\[
D^p_\mu(t_1, \mu_1) = D^p_{\mu_1}((\text{x} : \text{name} : ?\text{name}), \mu_1) \\
= (?x, \text{trg} : \text{title}, ?\text{name})
\]

Consequently, the output of Algorithm 2 is a graph pattern, having its triple patterns rewritten by their predicate part and is presented below:

\[
GP_p = (?x, \text{trg} : \text{title}, ?\text{name}) \ \text{AND} \ (\text{x} : \text{type} : \text{Science}) \\
\text{AND} \ (\text{x} : \text{type} : \text{Pocket})
\]

Then, Algorithm 1 uses Algorithm 2 in order to rewrite the triple patterns of \( GP_p \) by their object parts (step 4). All the triple patterns except of \( t_2 \) and \( t_3 \) remain the same, since they contain a SPARQL variable on their object part. The rewriting result of the triple patterns \( t_2 \) and \( t_3 \), is provided by invoking the function (6.8).

\[
D^o_\mu(t_2, \mu_2) = D^o_\mu((\text{x} : \text{type} : \text{Science}), \mu_2) \\
= (?x, \text{type} : \text{ComputerScience}) \\
\text{UNION} \ (?x, \text{type} : \text{Mathematics})
\]

\[
D^o_\mu(t_3, \mu_3) = D^o_\mu((\text{x} : \text{type} : \text{Pocket}), \mu_3) \\
= (?x, \text{type} : \text{Textbook}) \\
\text{AND} \ (?x, \text{size} : \text{?var}) \ \text{FILTER}(?\text{var} \leq 14)
\]

Consequently, the output of the second invocation of Algorithm 2 is a graph pattern, having its triple patterns rewritten by their object part and is presented below:
Finally, Algorithm 1 proceeds to step 5 (Algorithm 2 invocation) in order to rewrite the triple patterns of $GP_o$ by their subject parts. However, the resulted graph pattern $GP_s$ is the same with $GP_o$, since every triple pattern of $GP_o$ contains a SPARQL variable in its subject part.

\[
GP_s = (?x, trg : title, ?name) \text{ AND } \\
\left( (\?x, rdf : type, trg : \text{ComputerScience}) \text{ UNION } \\
(\?x, rdf : type, trg : \text{Mathematics}) \text{ AND } (?x, rdf : type, trg : \text{Textbook}) \right) \text{ AND } (?x, trg : size, ?var) \text{ FILTER(?var} \leq 14) 
\]

After removing any unnecessary brackets from $GP_s$ (step 6), the rewritten SPARQL query is provided by replacing the initial query’s graph pattern with $GP_s$. Consequently, the SPARQL query which will be posed over the target ontology of Figure 4.1, is presented below:

```
@PREFIX trg: <http://www.ontologies.com/TargetOntology.owl#>.
@PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.

SELECT ?name 
WHERE {?x trg:title ?name. 
  {?x rdf:type trg:ComputerScience} 
  UNION 
  {?x rdf:type trg:Mathematics} 
  ?x trg:size ?var. 
  FILTER(?var <= 14) }
```

**Example 8.4.** Consider the query posed over the source ontology of Figure 4.1: “Return the titles (at most 10) of the poetry or autobiography books written by Dante”. The SPARQL syntax of the source query is shown below:

```
@PREFIX src: <http://www.ontologies.com/SourceOntology.owl#>.
@PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.

SELECT ?name 
WHERE {?x src:title ?name. 
  {?x rdf:type src:Poetry} 
  UNION 
  {?x rdf:type src:Autobiography} 
  FILTER(?name = "Dante") }
```
SELECT ?name
WHERE {{?x rdf:type src:Poetry}
  UNION
  {?x rdf:type src:Autobiography}
?x src:name ?name.
FILTER regex(?author, "Dante")}
LIMIT 10

In order to rewrite the SPARQL query’s graph pattern \(GP_{in}\), Algorithm 1, as well as a set of predefined mappings \(M\) should be used. Let the available predefined mappings be the mappings \(\mu_1, \mu_2, \mu_3\) and \(\mu_4\) presented below. The inputs of Algorithm 1 are \(GP_{in}\) and \(M\).

\[
GP_{in} = \{(\?x, \text{rdf} : \text{type, src : Poetry}) \cup (\?x, \text{rdf} : \text{type, src : Autobiography})\} \\
\text{AND (\?x, src : author, ?author) AND (\?x, src : name, ?name)} \\
\text{FILTER (regex(?author,"Dante")})
\]

\[
M = \begin{cases} 
\mu_1 : \text{src : Poetry} \sqsubseteq \text{trg : Literature}, \\
\mu_2 : \text{src : Autobiography} \equiv \forall \text{trg : Biography}.(\text{trg : author} = \text{trg : topic}), \\
\mu_3 : \text{src : author} \equiv \text{trg : author} \circ \text{trg : name}, \\
\mu_4 : \text{src : name} \sqsubseteq \text{trg : title}
\end{cases}
\]

\(GP_{in}\) contains a \text{FILTER} operation which does not affect the rewriting procedure since the filter’s expression consists of SPARQL variables, literal constants and built-in functions. Consequently, the algorithm proceeds to the invocation of Algorithm 2 (step 3) in order to rewrite each triple pattern of \(GP_{in}\), using the mappings of the triple patterns’ predicate parts. The input of Algorithm 2 is the initial graph pattern \(GP_{in}\), as well as the set of mappings \(M\).

Algorithm 2 rewrites every basic graph pattern of \(GP_{in}\) triple pattern by triple pattern, using the mappings of the triple patterns’ predicate parts. The triple patterns \(t_1 = (\?x, \text{rdf} : \text{type, src : Poetry})\) and \(t_2 = (\?x, \text{rdf} : \text{type, src : Autobiography})\) remain the same after the rewriting process, since their predicate part consists of the RDF property \(\text{rdf} : \text{type}\). On the contrary, the rewriting result of the triple pattern \(t_3 = (\?x, \text{src : author, ?author})\), as well as the rewriting result of the triple pattern \(t_4 = (\?x, \text{src : name, ?name})\) are provided by invoking the function (6.11).
\[ D_{\varphi}^p(t_3, \mu_3) = D_{\varphi_{d p}}^p((?x, src : author, ?author), \mu_3) \]

\[ = (?x, trg : author, ?var) \text{ AND} \]
\[ (?var, trg : name, ?author) \]

\[ D_{\varphi}^p(t_4, \mu_4) = D_{\varphi_{d p}}^p((?x, src : name, ?name), \mu_4) \]

\[ = (?x, trg : title, ?name) \]

Consequently, the output of Algorithm 2 is a graph pattern, having its triple patterns rewritten by their predicate part and is presented below:

\[ GP_p = (\{?x, rdf : type, src : Poetry\} \cup (?x, rdf : type, src : Autobiography)) \]
\[ \text{AND} \ (?x, trg : author, ?var_1) \text{ AND} \ (?var_1, trg : name, ?author) \]
\[ \text{AND} \ (?x, trg : title, ?name) \text{ FILTER}(\text{regex(?author, "Dante")}) \]

Then, Algorithm 1 uses Algorithm 2 in order to rewrite the triple patterns of \( GP_p \) by their object parts (step 4). All the triple patterns except of \( t_1 \) and \( t_2 \) remain the same, since they contain a SPARQL variable on their object part. The rewriting result of the triple patterns \( t_1 \) and \( t_2 \), is provided by invoking the function (6.8).

\[ D_{\varphi}^o(t_1, \mu_1) = D_{\varphi_{d p}}^o((?x, rdf : type, src : Poetry), \mu_1) \]
\[ = (?x, rdf : type, trg : Literature) \]

\[ D_{\varphi}^o(t_2, \mu_2) = D_{\varphi_{d p}}^o((?x, rdf : type, src : Autobiography), \mu_2) \]
\[ = (?x, rdf : type, trg : Biography) \text{ AND} \]
\[ (?x, trg : author, ?var_2) \text{ AND} \]
\[ (?x, trg : topic, ?var_3) \text{ FILTER}(?var_2 = ?var_3) \]

Consequently, the output of second invocation of Algorithm 2 is a graph pattern, having its triple patterns rewritten by their object part and is presented below:

\[ GP_o = (\{?x, rdf : type, trg : Literature\} \cup (?x, rdf : type, trg : Biography)) \]
\[ \text{AND} \ (?x, trg : author, ?var_2) \text{ AND} \ (?x, trg : topic, ?var_3) \]
\[ \text{FILTER}(?var_2 = ?var_3) \] \text{ AND} \ (?x, trg : author, ?var_1) \]
\[ \text{AND} \ (?var_1, trg : name, ?author) \text{ AND} \ (?x, trg : title, ?name) \]
\[ \text{FILTER}(\text{regex(?author, "Dante")}) \]
Finally, Algorithm 1 proceeds to step 5 (Algorithm 2 invocation) in order to rewrite the triple patterns of $GP_o$ by their subject parts. However, the resulted graph pattern $GP_s$ is the same with $GP_o$, since every triple pattern of $GP_o$ contains a SPARQL variable in its subject part.

$$GP_s = \left( (?x, \text{rdf}: \text{type}, \text{trg}: \text{Literature}) \cup (?x, \text{rdf}: \text{type}, \text{trg}: \text{Biography}) \right) \land (\text{?x, trg: author, ?var}_2) \land (\text{?x, trg: topic, ?var}_3) \land (\text{FILTER}(\text{?var}_2 = \text{?var}_3)) \land (\text{?x, trg: author, ?var}_1) \land (\text{?var}_1, \text{trg: name, ?author}) \land (\text{?x, trg: title, ?name}) \land (\text{FILTER (regex(?author, "Dante") )})$$

After removing any unnecessary brackets from $GP_s$ (step 6), the rewritten SPARQL query is provided by replacing the initial query’s graph pattern with $GP_s$. Consequently, the SPARQL query which will be posed over the target ontology of Figure 4.1, is presented below:

```sparql
@PREFIX trg: <http://www.ontologies.com/TargetOntology.owl#>.
@PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.

SELECT ?name 
WHERE {{?x rdf:type trg:Literature} \cup {?x rdf:type trg:Biography. ?x trg:author ?var_2.
?x trg:title ?name. FILTER regex(?author, "Dante")}}
LIMIT 10
```
Chapter 9

Implementation

The SPARQL query rewriting methodology presented in this thesis has been implemented as part of a Semantic Query Mediation Prototype Infrastructure developed in the TUC-MUSIC Lab. The system has been implemented using Java 2SE as a software platform, and the Jena Software framework for SPARQL query parsing. The architecture of this infrastructure is shown in Figure 9.1 where many of the Mediator’s implementation details are not presented for simplicity reasons.

![Figure 9.1: System reference architecture.](image)

For each federated source, a dedicated Query Rewriting component is dynamically created by a Query Rewriting Factory. Such a component is able to rewrite an input
SPARQL query based on some predefined mappings. As a representation language for the mappings between two overlapping ontologies we use the language presented in Section 4.4.

During the system’s start-up each component is initialized with the mappings between the mediator’s global ontology and the (local) ontology used in the federated source for which this component is responsible.

At run time, when a SPARQL query is posed to the Mediator, it is processed, decomposed, and rewritten for each federated source by the corresponding query rewriting component. Afterwards, the rewritten queries are submitted (routed) to the federated sites for local evaluation. Finally, the returned results from the local sources (to which queries have been routed to) are merged, and optionally visualized (taking into account which part of the initial SPARQL query was answered by each resource) for presentation to the end users.
Chapter 10

Conclusion

The web of data is heterogeneous, distributed and structured. Querying mechanisms in this environment have to overcome the problems arising from the heterogeneous and distributed nature of data, and they have to allow the expression of complex and structured queries. The ontology mappings and SPARQL query mediation presented in this thesis aim to satisfy those requirements in the Semantic Web environment.

The mediator uses mappings between the global ontology of the mediator and the local ontologies of the federated knowledge bases. SPARQL queries of end users and applications which are posed over the mediator, are decomposed and rewritten in order to be submitted to the federated sources. The rewritten SPARQL queries are locally evaluated and the results are returned to the mediator. Two aspects of this system were discussed in this thesis:

- A formal model for describing executable ontology mappings (i.e. mappings which can be used in SPARQL query rewriting) that satisfy real-world requirements. We have presented a mapping model that allows the definition of a rich set of ontology mappings and we have shown real-world examples of its functionality.

- A complete set of SPARQL query rewriting functions and algorithms that allow SPARQL queries which are expressed based on the mediator’s global ontology to be rewritten in terms of the local ontologies. These functions are semantics preserving (i.e. preserve the mapping semantics).

Our current research focuses on the exploitation of N:M cardinality mappings by the query rewriting process, evaluating the system performance and exploiting advanced reasoning techniques during the query rewriting. Moreover, we aim to develop methodologies for the optimization of the query mediation process, as well as a graphical tool for mapping specification and automation of the mapping generation. Finally, this work is going to be integrated with the XS2OWL [40] and SPARQL2XQuery [6] frameworks (previous works of the TUC-MUSIC Lab), in order to allow access to heterogeneous web repositories.
Bibliography


Appendix A

Semantics of property relationships

A.1 Equivalence/subsumption between properties in OWL

Lemma A.1. The subsumption between two properties using the RDFS property rdfs: subPropertyOf implies subsumption between their domains, as well as subsumption between their ranges.

Proof. Let $p_1$, $p_2$ be object/datatype properties with domains $\text{domain}_{p_1}$, $\text{domain}_{p_2}$ and ranges $\text{range}_{p_1}$, $\text{range}_{p_2}$, respectively. The subsumption between these two properties ($p_1 \sqsubseteq p_2$) is interpreted in OWL by using the RDF triple $(p_1, \text{rdfs:subPropertyOf}, p_2)$. Considering that $\alpha \in \text{domain}_{p_1}$ and $b \in \text{range}_{p_1}$, each RDF triple of the form $(\alpha, p_1, b)$ implies the RDF triple $(\alpha, p_2, b)$.

Consequently, for the correspondences between the domains and ranges of the properties $p_1$ and $p_2$, we reach the following conclusions:

- $\forall x, [\text{domain}_{p_1}(x) \Rightarrow \text{domain}_{p_2}(x)]$, thus: $\text{domain}_{p_1} \sqsupseteq \text{domain}_{p_2}$
- $\forall y, [\text{range}_{p_1}(y) \Rightarrow \text{range}_{p_2}(y)]$, thus: $\text{range}_{p_1} \sqsupseteq \text{range}_{p_2}$

Lemma A.2. The equivalence between two properties using the OWL property owl: equivalentProperty implies equivalence between their domains, as well as equivalence between their ranges.

Proof. Let $p_1$, $p_2$ be object/datatype properties with domains $\text{domain}_{p_1}$, $\text{domain}_{p_2}$ and ranges $\text{range}_{p_1}$, $\text{range}_{p_2}$, respectively. The equivalence between these two properties ($p_1 \equiv p_2$) is interpreted in OWL by using the RDF triple $(p_1, \text{owl:equivalentProperty}, p_2)$, which indirectly implies the following RDF triples:
1. \((p_1, \text{rdf}s : \text{subPropertyOf}, p_2)\)

2. \((p_2, \text{rdf}s : \text{subPropertyOf}, p_1)\)

Let \(\alpha \in \text{domain}_{p_1}, \ b \in \text{range}_{p_1}, \ c \in \text{domain}_{p_2}\) and \(d \in \text{range}_{p_2}\), the RDF triples above, provide the following implications:

1. Each RDF triple of the form \((\alpha, p_1, b)\) implies the RDF triple \((\alpha, p_2, b)\).

2. Each RDF triple of the form \((c, p_2, d)\) implies the RDF triple \((c, p_1, d)\).

Consequently, for the correspondences between the domains and ranges of the properties \(p_1\) and \(p_2\), we reach the following conclusions:

- \(\forall x, [\text{domain}_{p_1}(x) \Leftrightarrow \text{domain}_{p_2}(x)], \text{thus: } \text{domain}_{p_1} \equiv \text{domain}_{p_2}\)
- \(\forall y, [\text{range}_{p_1}(y) \Leftrightarrow \text{range}_{p_2}(y)], \text{thus: } \text{range}_{p_1} \equiv \text{range}_{p_2}\)

### A.2 Equivalence/subsumption between properties in our framework

Among the mapping types defined in Section 4.3, there are correspondences between properties and property expressions. The statements below are directly implied by the semantics presented in Section 4.2.

**Lemma A.3.** An object property expression is an object property, having its domain and range dependent on the expression’s type. \(\square\)

**Lemma A.4.** A datatype property expression is a datatype property, having its domain and range dependent on the expression’s type. \(\square\)

Taking into consideration the OWL equivalence and subsumption semantics between properties (presented in Section A.1), as well as the fact that a property expression is actually a property having a domain and range, we reach the following conclusions which are also adapted in our framework:

- The subsumption \((\sqsubseteq, \sqsupseteq)\) between a property and a property expression implies subsumption between their domains, as well as subsumption between their ranges.

- The equivalence \((\equiv)\) between a property and a property expression implies equivalence between their domains, as well as equivalence between their ranges.
Appendix B

Preservation of semantics in Data Triple Pattern rewriting

In this appendix we provide the proofs of lemmas presented in Chapter 6 for the preservation of semantics in Data Triple Pattern rewriting. Refer to Table B.1 for the notation, as well as to Section 2.3.5 for the SPARQL graph pattern semantics, since they are used extensively in this appendix.

In what follows, let $DS_s$ and $DS_t$ be the RDF datasets of a source and a target ontology, respectively. Similarly, let $DS_m$ be the RDF dataset which is produced by merging the $DS_s$ and $DS_t$ datasets using a set of mappings $M$. Let $I$ be the interpretation that consists of the non-empty set $\Delta^I$, and contains the classes, the object/datatype properties and the individuals of the RDF dataset $DS_m$. The interpretation $I$ consists also of an interpretation function which assigns to every atomic concept $A$ a set $A^I \subseteq \Delta^I$, to every atomic role $B$ a binary relation $B^I \subseteq \Delta^I \times \Delta^I$ and to every individual $k$ an element $k^I \in \Delta^I$ (based on [2]).

Definition B.1 (Semantics preserving rewriting). Given a complete set (i.e. a set that contains every possible mapping) of sound (i.e. valid) mappings $M$ between $DS_s$ and $DS_t$, the rewriting step performed for a triple pattern $t$, based on a mapping $\mu \in M$, is semantics preserving if and only if the evaluation result of $t$ and the evaluation result of the rewritten graph pattern $gp'$ over $DS_m$, preserve the mapping semantics.

In other words, having a set $J = \text{var}(t)$ of SPARQL variables, the relationship $(\equiv, \sqsubseteq, \sqsupseteq)$ that holds for the mappings used in the rewriting process, should also hold between $[[t]]_{DS_m}$ and $[[gp']]_{DS_m}$ projected on $J$.

$$[[t]]_{DS_m} \; \text{rel} \; \pi_J \left( [[gp']]_{DS_m} \right), \; \text{rel} := \equiv | \sqsubseteq | \sqsupseteq$$  \hspace{1cm} (B.1)
Table B.1: The notation which is used for the Data triple pattern rewriting proofs.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\omega$</td>
<td>A graph pattern solution $\omega: V \rightarrow (I \cup B \cup L)$.</td>
</tr>
<tr>
<td>$\text{dom}(\omega)$</td>
<td>Domain of a graph pattern solution $\omega$ (subset of $V$).</td>
</tr>
<tr>
<td>$\omega(t)$</td>
<td>The triple obtained by replacing the variables in triple pattern $t$ according to a graph pattern solution $\omega$ (abusing notation).</td>
</tr>
<tr>
<td>$\omega^s(t)$</td>
<td>The subject part of the triple obtained by replacing the variables in triple pattern $t$ according to a graph pattern solution $\omega$.</td>
</tr>
<tr>
<td>$\omega^p(t)$</td>
<td>The predicate part of the triple obtained by replacing the variables in triple pattern $t$ according to a graph pattern solution $\omega$.</td>
</tr>
<tr>
<td>$\omega^o(t)$</td>
<td>The object part of the triple obtained by replacing the variables in triple pattern $t$ according to a graph pattern solution $\omega$.</td>
</tr>
<tr>
<td>$\omega \models R$</td>
<td>A graph pattern solution $\omega$ satisfies a built-in condition $R$.</td>
</tr>
<tr>
<td>$[[\cdot]]$</td>
<td>Graph pattern evaluation function.</td>
</tr>
<tr>
<td>$\text{var}(GP)$</td>
<td>The variables of a graph pattern $GP$.</td>
</tr>
<tr>
<td>$\bowtie$</td>
<td>Graph pattern solution-based join.</td>
</tr>
<tr>
<td>$\bowtie_l$</td>
<td>Graph pattern solution-based left outer join.</td>
</tr>
<tr>
<td>$\setminus$</td>
<td>Graph pattern solution-based difference.</td>
</tr>
<tr>
<td>$\pi{\ldots}$</td>
<td>Graph pattern solution-based projection.</td>
</tr>
<tr>
<td>$\cup$</td>
<td>Graph pattern solution-based union.</td>
</tr>
<tr>
<td>$\cap$</td>
<td>Set intersection.</td>
</tr>
<tr>
<td>AND, OPT, UNION, FILTER</td>
<td>SPARQL graph pattern operators.</td>
</tr>
<tr>
<td>$\neg$, $\lor$, $\land$</td>
<td>Logical not, or, and.</td>
</tr>
<tr>
<td>$=, \leq, \geq, &lt;, &gt;$</td>
<td>Inequality/equality operators.</td>
</tr>
</tbody>
</table>

**Proof Overview.** The proofs presented in this appendix refer to Data Triple Patterns and follow a common approach. We consider mappings containing equivalence relationship ($\equiv$) and we do not provide the proofs for the other mapping types since the approach is very similar for all types.

Let $t$ be the initial triple pattern, $gp'$ be the rewritten graph pattern and $J$ the set of SPARQL variables appearing in $t$. First, we use the mapping semantics in order to show that every graph pattern solution of $t$ over the RDF dataset $DS_m$ is a graph pattern...
solution of \( gp' \) over \( DS_m \), for the common graph pattern solution domain \( J = \text{var}(t) \), inferring that:

\[
[[t]]_{DS_m} \subseteq \pi_J([[gp']_{DS_m})]
\]  

(B.3)

Then, we show that every graph pattern solution of \( gp' \) over the RDF dataset \( DS_m \) is a graph pattern solution of \( t \) over \( DS_m \), for the common graph pattern solution domain \( J \), inferring that:

\[
[[t]]_{DS_m} \supseteq \pi_J([[gp']_{DS_m})]
\]  

(B.4)

From (B.3) and (B.4) we derive that \( [[t]]_{DS_m} \equiv \pi_J([[gp']_{DS_m}) \). Considering that the mapping used for the rewriting process has the same relationship (equivalence), we conclude the proof. Similarly, for mapping types containing subsumption relationships (\( \sqsubseteq, \sqsupseteq \)), we reach either to (B.3) or to (B.4) using the mapping semantics, proving that the rewriting step is semantics preserving (i.e. preserves the mappings semantics).

### B.1 Proof of Lemma 6.1

In this section, we prove that the rewriting step performed for a Data Triple Pattern, based on a mapping of its subject part, is semantics preserving. According to Lemma 6.1, the only case that we examine concerns individuals appearing in the triple pattern’s subject part.

Let \( i_s \) be an individual from the source ontology, \( t = (i_s, \text{predicate}, \text{object}) \) be a Data Triple Pattern and \( J \) be the set of SPARQL variables appearing in \( t \). The evaluation of the triple pattern \( t \) over the RDF dataset \( DS_m \) is presented below:

\[
[[t]]_{DS_m} = [[[i_s, \text{predicate}, \text{object}]_{DS_m}]
\]

We consider the following case:

1. Let \( i_t \) be an individual from the target ontology. Having a mapping \( \mu : i_s \equiv i_t \) (i.e. \( i_s^T \equiv i_t^T \)) the rewritten (based on \( t \)’s subject part) graph pattern \( gp' \) and its evaluation over the RDF dataset \( DS_m \) are the following:

\[
gp' = \mathcal{D}'_s(t, \mu) = (i_t, \text{predicate}, \text{object})
\]

\[
[[gp']_{DS_m} = [[[i_t, \text{predicate}, \text{object}]_{DS_m}]
\]
We consider two premises:

(a) \( \forall \omega \in [[t]]_{DS_m} : \exists x, \exists y, \) such that \((i_s, x, y) \in DS_m, \omega^p(t) = x \) and \( \omega^o(t) = y. \) Moreover, since \( i_s \equiv i_t \) then \((i_t, x, y) \in DS_m, \) inferring that \( \omega \in \pi_J ([[gp']])_{DS_m}. \)

(b) \( \forall \omega \in \pi_J ([[gp']])_{DS_m} : \exists x, \exists y, \) such that \((i_t, x, y) \in DS_m, \omega^p(gp') = x \) and \( \omega^o(gp') = y. \) Moreover, since \( i_s \equiv i_t \) then \((i_s, x, y) \in DS_m, \) inferring that \( \omega \in [[t]]_{DS_m}. \)

From the two premises above, we conclude that:

\[ [[t]]_{DS_m} \equiv \pi_J ([[gp']])_{DS_m} \]

This concludes the proof that the rewriting step of a Data Triple Pattern, based on a mapping of its subject part, is semantics preserving.

### B.2 Proof of Lemma 6.2

In this section, we prove that the rewriting step performed for a Data Triple Pattern, based on a mapping of its object part, is semantics preserving. According to Lemma 6.2, the only cases that we examine concern individuals and classes appearing in the triple pattern’s object part.

To begin with, we prove that the rewriting step performed for a Data Triple Pattern, based on an individual mapping of its object part, is semantics preserving. Let \( i_s \) be an individual from the source ontology, \( t = (subject, predicate, i_s) \) be a Data Triple Pattern and \( J \) be the set of SPARQL variables appearing in \( t. \) The evaluation of the triple pattern \( t \) over the RDF dataset \( DS_m \) is presented below:

\[ [[t]]_{DS_m} = [[[subject, predicate, i_s]]]_{DS_m} \]

We consider the following case:

1. Let \( i_t \) be an individual from the target ontology. Having a mapping \( \mu : i_s \equiv i_t \) (i.e. \( \mu_i = \mu_i \)), the rewritten (based on \( t \)’s object part) graph pattern \( gp' \) and its evaluation over the RDF dataset \( DS_m \) are the following:

\[ gp' = D^p_o(t, \mu) = (subject, predicate, i_t) \]

\[ [[[gp']]_{DS_m} = [[[subject, predicate, i_t]]]_{DS_m} \]
We consider two premises:

(a) \( \forall \omega \in [[t]]_{DS_m} : \exists x, \exists y, \text{ such that } (x, y, i_s) \in DS_m, \omega^s(t) = x \) and \( \omega^p(t) = y \). Moreover, since \( i_s \equiv i_t \) then \( (x, y, i_t) \in DS_m \), inferring that \( \omega \in \pi_J([[gp']])_{DS_m} \).

(b) \( \forall \omega \in \pi_J([[gp']])_{DS_m} : \exists x, \exists y, \text{ such that } (x, y, i_t) \in DS_m, \omega^s(gp') = x \) and \( \omega^p(gp') = y \). Moreover, since \( i_s \equiv i_t \) then \( (x, y, i_s) \in DS_m \), inferring that \( \omega \in [[t]]_{DS_m} \).

From the two premises above, we conclude that:

\[ [[t]]_{DS_m} \equiv \pi_J([[gp']])_{DS_m} \]

This concludes the proof that the rewriting step of a Data Triple Pattern, based on an individual mapping of its object part, is semantics preserving.

Similarly, we prove that the rewriting step performed for a Data Triple Pattern, based on a class mapping of its object part, is semantics preserving. Let \( c_s \) be a class from the source ontology, \( t = (\text{subject}, \text{rdf} : \text{type}, c_s) \) be a Data Triple Pattern and \( J \) be the set of SPARQL variables appearing in \( t \). The evaluation of the triple pattern \( t \) over the RDF dataset \( DS_m \) is presented below:

\[ [[t]]_{DS_m} = [[[\text{subject}, \text{rdf} : \text{type}, c_s]]]_{DS_m} \]

For the different types of class mappings, we consider the following cases:

1. Let \( c_t \) be a class from the target ontology. Having a mapping \( \mu : c_s \equiv c_t \) (i.e. \( c_t^I = c_s^I \)), the rewritten (based on \( t \)'s object part) graph pattern \( gp' \) and its evaluation over the RDF dataset \( DS_m \) are the following:

\[ gp' = D^\mu_c(t, \mu) = (\text{subject}, \text{rdf} : \text{type}, c_t) \]

\[ [[gp']]_{DS_m} = [[[\text{subject}, \text{rdf} : \text{type}, c_t]]]_{DS_m} \]

We consider two premises:

(a) \( \forall \omega \in [[t]]_{DS_m} : \exists x, \text{ such that } (x, \text{rdf} : \text{type}, c_s) \in DS_m \) and \( \omega^s(t) = x \). Thus, \( x \in c_s^I \) and since \( c_s^I = c_t^I \) then \( (x, \text{rdf} : \text{type}, c_t) \in DS_m \), inferring that \( \omega \in \pi_J([[gp']]_{DS_m}) \).
Let \( c \) (i.e. \( c = c_1 \cap c_2 \)), the rewritten (based on \( t \)'s object part) graph pattern \( gp' \) and its evaluation over the RDF dataset \( DS_m \) are the following:

\[
gp' = \mathcal{D}^\ell(t, \mu) = (\text{subject}, \text{rdf} : \text{type}, c_1) \cup (\text{subject}, \text{rdf} : \text{type}, c_2)
\]

\[
[[gp']]_{DS_m} = [[[\text{subject}, \text{rdf} : \text{type}, c_1]]]_{DS_m} \cup [[[\text{subject}, \text{rdf} : \text{type}, c_2]]]_{DS_m}
\]

We consider two premises:

(a) \( \forall \omega \in [[t]]_{DS_m} : \exists x, \) such that \( (x, \text{rdf} : \text{type}, c_1) \in DS_m \) and \( \omega^s(t) = x \). Thus, \( x \in c_1^T \) and \( c_1^T = c_1 \cap c_2 \), then \( (x, \text{rdf} : \text{type}, c_1) \in DS_m \). From the two premises above, we conclude that:

\[
[[t]]_{DS_m} \equiv \pi_\ell([[gp']]_{DS_m})
\]

2. Let \( c_1 \) and \( c_2 \) be classes from the target ontology. Having a mapping \( \mu : c_4 \equiv c_1 \cap c_2 \) (i.e. \( c_1^T = c_1 \cap c_2 \)), the rewritten (based on \( t \)'s object part) graph pattern \( gp' \) and its evaluation over the RDF dataset \( DS_m \) are the following:

\[
gp' = \mathcal{D}^\ell(t, \mu) = (\text{subject}, \text{rdf} : \text{type}, c_1) \cup (\text{subject}, \text{rdf} : \text{type}, c_2)
\]

\[
[[gp']]_{DS_m} = [[[\text{subject}, \text{rdf} : \text{type}, c_1]]]_{DS_m} \cup [[[\text{subject}, \text{rdf} : \text{type}, c_2]]]_{DS_m}
\]

We consider two premises:

(a) \( \forall \omega \in [[t]]_{DS_m} : \exists x, \) such that \( (x, \text{rdf} : \text{type}, c_1) \in DS_m \) and \( \omega^s(t) = x \). Thus, \( x \in c_1^T \) and \( c_1^T = c_1 \cap c_2 \), then \( (x, \text{rdf} : \text{type}, c_1) \in DS_m \). From the two premises above, we conclude that:

\[
[[t]]_{DS_m} \equiv \pi_\ell([[gp']]_{DS_m})
\]

3. Let \( c_1 \) and \( c_2 \) be classes from the target ontology. Having a mapping \( \mu : c_4 \equiv c_1 \cap c_2 \) (i.e. \( c_1^T = c_1 \cap c_2 \)), the rewritten (based on \( t \)'s object part) graph pattern \( gp' \) and its evaluation over the RDF dataset \( DS_m \) are the following:

\[
gp' = \mathcal{D}^\ell(t, \mu) = (\text{subject}, \text{rdf} : \text{type}, c_1) \cup (\text{subject}, \text{rdf} : \text{type}, c_2)
\]
We consider two premises:

(a) \( \forall \omega \in [[t]]_{DS_m} : \exists x, \text{ such that } (x, \text{rdf}: \text{type}, c_s) \in DS_m \text{ and } \omega^s(t) = x \). Thus, 
    \( x \in c^s_x \) and since \( c^s_x = c^s_1 \cap c^s_2 \) then \( (x, \text{rdf}: \text{type}, c_{s1}) \in DS_m \) and \( (x, \text{rdf}: \text{type}, c_{s2}) \in DS_m \), inferring that \( \omega \in \pi_j ([[gp']_{DS_m}]) \).

(b) \( \forall \omega \in \pi_j ([[gp']_{DS_m}]) : \exists x, \text{ such that } (x, \text{rdf}: \text{type}, c_s) \in DS_m, (x, \text{rdf}: \text{type}, c_{s2}) \in DS_m \text{ and } \omega^s(t_1^s) = \omega^s(t_2^s) = x \). Thus, \( x \in c^s_1 \cap c^s_2 \) and since \( c^s_x = c^s_1 \cap c^s_2 \) then \( (x, \text{rdf}: \text{type}, c_s) \in DS_m \), inferring that \( \omega \in [[t]]_{DS_m} \).

From the two premises above, we conclude that:

\[
[[t]]_{DS_m} \equiv \pi_j ([[gp']_{DS_m})
\]

4. Let \( c_t \) be a class, \( op_t \) be an object property from the target ontology, \( v_{op} \) be an individual and \( \text{op} \in \{\neq, =\} \). Having a mapping \( \mu : c_s \rightarrow c_t, (\text{op}_t \text{op} v_{op}) \) (i.e. \( c_t^s = \{ \alpha \in c^s_t | \exists b, (\alpha, b) \in \text{op}^s_t \land b \text{op} v_{op} \} \)), the rewritten (based on \( t \)’s object part) graph pattern \( gp' \) and its evaluation over the RDF dataset \( DS_m \) are the following:

\[
gp' = \mathcal{D}_0^v(t, \mu) = (\text{subject}, \text{rdf} : \text{type}, c_t) \text{ AND } (\text{subject}, \text{op}_t, \text{?var}) \text{ FILTER}(?\text{var} \text{op} v_{op})
\]

\[
[[gp']_{DS_m} = [[[\text{subject}, \text{rdf} : \text{type}, c_t] \text{ AND } (\text{subject}, \text{op}_t, \text{?var}) \text{ FILTER}(?\text{var} \text{op} v_{op})]]_{DS_m}
\]

\[
= \{ \omega \in [[[\text{subject}, \text{rdf} : \text{type}, c_t]]_{DS_m} \uplus [[[\text{subject}, \text{op}_t, \text{?var}]]]_{DS_m})
\mid \omega \models \text{?var} \text{op} v_{op} \}
\]

\[
= \{ \omega \in [[[t_1]]_{DS_m} \uplus [[[t_2]]]_{DS_m}) \mid \omega \models \text{?var} \text{op} v_{op} \}
\]

Let \( \mathcal{L} = \{ \alpha \in c^s_t | \exists b, (\alpha, b) \in \text{op}^s_t \land b \text{op} v_{op} \} \). We consider two premises:

(a) \( \forall \omega \in [[t]]_{DS_m} : \exists x, \text{ such that } (x, \text{rdf}: \text{type}, c_s) \in DS_m \text{ and } \omega^s(t) = x \). Thus, 
    \( x \in c^s_x \) and since \( c^s_x = \mathcal{L} \) then \( (x, \text{rdf}: \text{type}, c_s) \in DS_m, \exists y \text{ such that } (x, \text{op}_t, y) \in DS_m \text{ and } y \text{op} v_{op}, \inferring that } \omega \in \pi_j ([[gp']_{DS_m}) \).

∀ \omega \in \pi_\varphi([[gp']]|_{DS_m}) : \exists x, such that (x, rdf : type, c_i) \in DS_m, \exists y such that (x, op, y) \in DS_m, y \not\in v_{dp} \land \omega^*(t'_1) = \omega^*(t'_2) = x. Thus, \omega \in |t|_{DS_m}.

From the two premises above, we conclude that:

\[
[[t]|_{DS_m} \equiv \pi_\varphi([[gp']]|_{DS_m})
\]

\[\]

5. Let \( c_i \) be a class, \( dp_t \) be a datatype property from the target ontology, \( v_{dp} \) be a data value and \( cp \in \{\neq, =, >, <, \geq, \leq\} \). Having a mapping \( \mu : c_s \rightarrow c_t(dp_t \ circ v_{dp}) \) (i.e. \( c^T_s = \{a \in c^T_t \mid \exists b. (a, b) \in dp^T_t \land b \ circ v_{dp}\} \), the rewritten (based on \( t \)'s object part) graph pattern \( gp' \) and its evaluation over the RDF dataset \( DS_m \) are the following:

\[
gp' = \mathcal{D}_c(t, \mu) = (subject, rdf : type, c_i) \land (subject, dp_t, ?var) \land \text{FILTER(?var \ circ v_{dp})}
\]

\[
[[gp']|_{DS_m} = \{[\text{subject, rdf : type, c_i}] \land [\text{subject, dp_t, ?var}] \land \text{FILTER(?var \ circ v_{dp})}|_{DS_m}
\]

\[
\{\omega \in \{(subject, rdf : type, c_i)|_{DS_m} \land (subject, dp_t, ?var)|_{DS_m}\} \mid \omega \models ?var \ circ v_{dp}\}
\]

\[
\{\omega \in \{(subject, rdf : type, c_i)|_{DS_m} \land (subject, dp_t, ?var)|_{DS_m}\} \mid \omega \models ?var \ circ v_{dp}\}
\]

Let \( \mathcal{L} = \{a \in c^T_t \mid \exists b. (a, b) \in dp^T_t \land b \ circ v_{dp}\} \). We consider two premises:

(a) \( \forall \omega \in |t|_{DS_m} : \exists x, such that (x, rdf : type, c_i) \in DS_m \land \omega^*(t) = x. Thus, x \in c^T_s \) and since \( c^T_s = \mathcal{L} \) then \( (x, rdf : type, c_i) \in DS_m \), \( \exists y \ such that (x, dp_t, y) \in DS_m \) and \( y \ circ v_{dp} \), inferring that \( \omega \in \pi_\varphi([[gp']]|_{DS_m}) \).

(b) \( \forall \omega \in \pi_\varphi([[gp']]|_{DS_m}) : \exists x, such that (x, rdf : type, c_i) \in DS_m, \exists y such that (x, dp_t, y) \in DS_m, y \ circ v_{dp} \land \omega^*(t'_1) = \omega^*(t'_2) = x. Thus, x \in \mathcal{L} \) and since \( c^T_s = \mathcal{L} \) then \( (x, rdf : type, c_i) \in DS_m \), inferring that \( \omega \in |t|_{DS_m} \).

From the two premises above, we conclude that:

\[
[[t]|_{DS_m} \equiv \pi_\varphi([[gp']]|_{DS_m})
\]
6. Let $c_t$ be a class, $op_{t1}$, $op_{t2}$ be object properties from the target ontology and $\mathcal{CP} \in \{\neq, =\}$. Having a mapping $\mu : c_s \rightarrow c_t.(op_{t1} \mathcal{CP} op_{t2})$ (i.e. $c^T_s = \{\alpha \in c^T_t \mid \exists b, \exists c. (\alpha, b) \in op_{t1}^T \land (\alpha, c) \in op_{t2}^T \land b \mathcal{CP} c\}$), the rewritten (based on $t$'s object part) graph pattern $gp'$ and its evaluation over the RDF dataset $DS_m$ are the following:

$$gp' = D^\mu_s(t, \mu) = (subject, rdf : type, c_t) \text{ AND } (subject, op_{t1}, ?var_1) \text{ AND } (subject, op_{t2}, ?var_2)$$

$$[[gp']]_{DS_m} = [[[subject, rdf : type, c_t]] \text{ AND } (subject, op_{t1}, ?var_1) \text{ AND } (subject, op_{t2}, ?var_2)]_{DS_m}$$

$$= \{\omega \in [[[subject, rdf : type, c_t]]_{DS_m} \uplus [[[subject, op_{t1}, ?var_1]]_{DS_m} \uplus [[[subject, op_{t2}, ?var_2]]_{DS_m}] \mid \omega \models ?var_1 \mathcal{CP} ?var_2\}$$

Let $L = \{\alpha \in c^T_t \mid \exists b, \exists c. (\alpha, b) \in op_{t1}^T \land (\alpha, c) \in op_{t2}^T \land b \mathcal{CP} c\}$. We consider two premises:

(a) $\forall \omega \in [[t]]_{DS_m} : \exists x$, such that $(x, rdf : type, c_t) \in DS_m \text{ and } \omega^s(t) = x$. Thus, $x \in c^T_t$ and since $c^T_s = L$ then $(x, rdf : type, c_t) \in DS_m$, $\exists y$ such that $(x, op_{t1}, y) \in DS_m$, $\exists z$ such that $(x, op_{t2}, z) \in DS_m$ and $y \mathcal{CP} z$, inferring that $\omega \models \pi_x([[gp']]_{DS_m})$.

(b) $\forall \omega \in \pi_x([[gp']]_{DS_m}) : \exists x$, such that $(x, rdf : type, c_t) \in DS_m$, $\exists y$ such that $(x, op_{t1}, y) \in DS_m$, $\exists z$ such that $(x, op_{t2}, z) \in DS_m$, $y \mathcal{CP} z$ and $\omega^s(t'_1) = \omega^s(t'_2) = \omega^s(t'_3) = x$. Thus, $x \in L$ and since $c^T_s = L$ then $(x, rdf : type, c_t) \in DS_m$, inferring that $\omega \models [[t]]_{DS_m}$.

From the two premises above, we conclude that:

$$[[t]]_{DS_m} \equiv \pi_x([[gp']]_{DS_m})$$

7. Let $c_t$ be a class, $dp_{t1}$, $dp_{t2}$ be datatype properties from the target ontology and $\mathcal{CP} \in \{\neq, =\}$. Having a mapping $\mu : c_s \rightarrow c_t.(dp_{t1} \mathcal{CP} dp_{t2})$ (i.e. $c^T_s = \{\alpha \in c^T_t \mid \exists b, \exists c. (\alpha, b) \in dp_{t1}^T \land (\alpha, c) \in dp_{t2}^T \land b \mathcal{CP} c\}$), the rewritten (based on $t$'s object part) graph pattern $gp'$ and its evaluation over the RDF dataset $DS_m$ are the following:
∀ based on an object property mapping of its predicate part, is semantics preserving. Let pattern’s predicate part.

In this section, we prove that the rewriting step performed for a Data Triple Pattern, based on a mapping of its object part, is semantics preserving. According to Lemma 6.3, the only cases that we examine concern object and datatype properties appearing in the triple pattern’s predicate part.

B.3 Proof of Lemma 6.3

In this section, we prove that the rewriting step performed for a Data Triple Pattern, based on a mapping of its predicate part, is semantics preserving. According to Lemma 6.3, the only cases that we examine concern object and datatype properties appearing in the triple pattern’s predicate part.

To begin with, we prove that the rewriting step performed for a Data Triple Pattern, based on an object property mapping of its predicate part, is semantics preserving. Let
$op_s$ be an object property from the source ontology, $t = (subject, op_s, object)$ be a Data Triple Pattern and $J$ be the set of SPARQL variables appearing in $t$. The evaluation of the triple pattern $t$ over the RDF dataset $DS_m$ is presented below:

$$[[t]]_{DS_m} = [[[subject, op_s, object]]]_{DS_m}$$

For the different types of object property mappings, we consider the following cases:

1. Let $op_t$ be an object property from the target ontology. Having a mapping $\mu : op_s \equiv op_t$ (i.e. $op_s^T = op_t^T$), the rewritten (based on $t$’s predicate part) graph pattern $gp'$ and its evaluation over the RDF dataset $DS_m$ are the following:

$$gp' = D_{op_s}(t, \mu) = (subject, op_t, object)$$

$$[[gp']]_{DS_m} = [[[subject, op_t, object]]]_{DS_m}$$

We consider two premises:

(a) $\forall \omega \in [[t]]_{DS_m} : \exists x, \exists y, \text{ such that } (x, op_s, y) \in DS_m, \omega^a(t) = x \text{ and } \omega^o(t) = y$. Thus, $(x, y) \in op_s^T$ and since $op_s^T = op_t^T$ then $(x, op_t, y) \in DS_m$, inferring that $\omega \in \pi_J([[[gp']]]_{DS_m})$.

(b) $\forall \omega \in \pi_J([[[gp']]_{DS_m}) : \exists x, \exists y, \text{ such that } (x, op_t, y) \in DS_m, \omega^a(gp') = x \text{ and } \omega^o(gp') = y$. Thus, $(x, y) \in op_t^T$ and since $op_s^T = op_t^T$ then $(x, op_s, y) \in DS_m$, inferring that $\omega \in [[t]]_{DS_m}$.

From the two premises above, we conclude that:

$$[[t]]_{DS_m} \equiv \pi_J([[[gp']]_{DS_m})$$

2. Let $op_{t1}$ and $op_{t2}$ be object properties from the target ontology. Having a mapping $\mu : op_s \equiv op_{t1} \circ op_{t2}$ (i.e. $op_s^T = \{((\alpha, c) \mid \exists b. (\alpha, b) \in op_{t1}^T \land (b, c) \in op_{t2}^T)\}$), the rewritten (based on $t$’s predicate part) graph pattern $gp'$ and its evaluation over the RDF dataset $DS_m$ are the following:

$$gp' = D_{op_s}(t, \mu) = (subject, op_{t1}, ?var) \text{ AND } (?var, op_{t2}, object)$$
\[ [[gp']]_{DS_m} = [[[\text{subject}, \text{op}_1, \text{?} \text{var}] \text{ AND } (?\text{var}, \text{op}_2, \text{object})]]_{DS_m} \]
\[ = [[[\text{subject}, \text{op}_1, ?\text{var}]])_{DS_m} \setminus_1 [[[?\text{var}, \text{op}_2, \text{object})]]_{DS_m} \]
\[ = [[t'_1]]_{DS_m} \setminus_1 [[t'_2]]_{DS_m} \]

Let \( \mathcal{L} = \{(a, c) \mid \exists b. (a, b) \in \text{op}_1 \land (b, c) \in \text{op}_2\} \). We consider two premises:

(a) \( \forall \omega \in [[t]]_{DS_m} : \exists x, \exists y, \text{ such that } (x, \text{op}_s, y) \in DS_m, \omega^s(t) = x \text{ and } \omega^o(t) = y. \)

Thus, \((x, y) \in \text{op}_s^x \text{ and since op}_s^x = \mathcal{L} \text{ then } \exists z \text{ such that } (x, \text{op}_1, z) \in DS_m \text{ and } (z, \text{op}_2, y) \in DS_m, \text{ inferring that } \omega \in \pi_{\mathcal{L}}([[gp']]_{DS_m}) \).

(b) \( \forall \omega \in \pi_{\mathcal{L}}([[gp']]_{DS_m}) : \exists x, \exists y, \exists z, \text{ such that } (x, \text{op}_1, z) \in DS_m, (z, \text{op}_2, y) \in DS_m, \omega^s(t'_1) = x, \omega^o(t'_1) = \omega^s(t'_2) = z \text{ and } \omega^o(t'_2) = y. \)

Thus, \((x, y) \in \mathcal{L} \text{ and since op}_s^x = \mathcal{L} \text{ then } (x, \text{op}_s, y) \in DS_m, \text{ inferring that } \omega \in [[t]]_{DS_m} \).

From the two premises above, we conclude that:
\[ [[t]]_{DS_m} \equiv \pi_{\mathcal{L}}([[gp']]_{DS_m}) \]

3. Let \( \text{op}_1 \) and \( \text{op}_2 \) be object properties from the target ontology. Having a mapping \( \mu : \text{op}_s \equiv \text{op}_1 \cup \text{op}_2 \) (i.e. \( \text{op}_s^x = \text{op}_1^x \cup \text{op}_2^x \)), the rewritten (based on \( t \)’s predicate part) graph pattern \( gp \) and its evaluation over the RDF dataset \( DS_m \) are the following:
\[ gp' = D_{\text{op}}(t, \mu) = (\text{subject}, \text{op}_1, \text{object}) \text{ UNION } (\text{subject}, \text{op}_2, \text{object}) \]
\[ [[gp']]_{DS_m} = [[[\text{subject}, \text{op}_1, \text{object}] \text{ UNION } (\text{subject}, \text{op}_2, \text{object})]]_{DS_m} \]
\[ = [[[\text{subject}, \text{op}_1, \text{object}]])_{DS_m} \cup [[[\text{subject}, \text{op}_2, \text{object})]]_{DS_m} \]
\[ = [[t'_1]]_{DS_m} \cup [[t'_2]]_{DS_m} \]

We consider two premises:

(a) \( \forall \omega \in [[t]]_{DS_m} : \exists x, \exists y, \text{ such that } (x, \text{op}_s, y) \in DS_m, \omega^s(t) = x \text{ and } \omega^o(t) = y. \)

Thus, \((x, y) \in \text{op}_s^x \text{ and since op}_s^x = \text{op}_1 \cup \text{op}_2 \text{ then } (x, \text{op}_1, y) \in DS_m \text{ or } (x, \text{op}_2, y) \in DS_m, \text{ inferring that } \omega \in \pi_{\mathcal{L}}([[gp']]_{DS_m}). \)

(b) \( \forall \omega \in \pi_{\mathcal{L}}([[gp']]_{DS_m}) : \exists x, \exists y, \text{ such that } (x, \text{op}_1, y) \in DS_m \text{ or } (x, \text{op}_2, y) \in DS_m, \omega^s(t'_1) = x \text{ or } \omega^s(t'_2) = x, \text{ and } \omega^o(t'_1) = y \text{ or } \omega^o(t'_2) = y. \)

Thus, \((x, y) \in \text{op}_1 \cup \text{op}_2 \text{ and since op}_s^x = \text{op}_1 \cup \text{op}_2 \text{ then } (x, \text{op}_s, y) \in DS_m, \text{ inferring that } \omega \in [[t]]_{DS_m} \).
From the two premises above, we conclude that:

$$[[t]]_{DS_m} \equiv \pi_J ([[g]\prime])_{DS_m}$$

4. Let $op_1$ and $op_2$ be object properties from the target ontology. Having a mapping $\mu : op_s \equiv op_1 \cap op_2$ (i.e. $op_s' = op_1' \cap op_2'$), the rewritten (based on $t$’s predicate part) graph pattern $g\prime$ and its evaluation over the RDF dataset $DS_m$ are the following:

$$g\prime = D_{op}(t, \mu) = (subject, op_1, object) \text{ AND } (subject, op_2, object)$$

$$[[g]\prime]_{DS_m} = [[[subject, op_1, object]]]_{DS_m} \text{ AND } [[[subject, op_2, object]]]_{DS_m}$$

$$= [[[subject, op_1, object]]]_{DS_m} \uplus [[[subject, op_2, object]]]_{DS_m}$$

$$= [[[t_0^1]]_{DS_m} \uplus [[[t_0^2]]_{DS_m}}$$

We consider two premises:

(a) $\forall \omega \in [[t]]_{DS_m} : \exists x, \exists y$, such that $(x, op_s, y) \in DS_m$, $\omega^s(t) = x$ and $\omega^o(t) = y$. Thus, $(x, y) \in op_s$ and since $op_s' = op_1' \cap op_2'$ then $(x, op_1, y) \in DS_m$ and $(x, op_2, y) \in DS_m$, inferring that $\omega \in \pi_J ([[g]\prime])_{DS_m}$.

(b) $\forall \omega \in \pi_J ([[g]\prime])_{DS_m} : \exists x, \exists y$, such that $(x, op_1, y) \in DS_m$, $(x, op_2, y) \in DS_m$, $\omega^s(t_0^1) = \omega^s(t_0^2) = x$ and $\omega^o(t_0^1) = \omega^o(t_0^2) = y$. Thus, $(x, y) \in op_1' \cap op_2'$ and since $op_s' = op_1' \cap op_2'$ then $(x, op_s, y) \in DS_m$, inferring that $\omega \in [[t]]_{DS_m}$.

From the two premises above, we conclude that:

$$[[t]]_{DS_m} \equiv \pi_J ([[g]\prime])_{DS_m}$$

5. Let $op_t$ be an object property from the target ontology. Having a mapping $\mu : op_s \equiv inv(op_t)$ (i.e. $op_s' = \{(b, a) \mid (\alpha, b) \in op_t'\}$), the rewritten (based on $t$’s predicate part) graph pattern $g\prime$ and its evaluation over the RDF dataset $DS_m$ are the following:

$$g\prime = D_{op}(t, \mu) = (object, op_t, subject)$$

$$[[g]\prime]_{DS_m} = [[[object, op_t, subject]]]_{DS_m}$$
Let $\mathcal{L} = \{(b, \alpha) \mid (\alpha, b) \in \text{op}_t^\tau\}$. We consider two premises:

(a) $\forall \omega \in [[t]]_{DS_m} : \exists x, \exists y,$ such that $(x, \text{op}_s, y) \in DS_m$, $\omega^\tau(t) = x$ and $\omega^\alpha(t) = y$. Thus, $(x, y) \in \text{op}_s^\tau$ and since $\text{op}_s^\tau = \mathcal{L}$ then $(y, \text{op}_t, x) \in DS_m$, inferring that $\omega \in \pi_\mathcal{J}([[\text{gp}'], DS_m])$.

(b) $\forall \omega \in \pi_\mathcal{J}([[\text{gp}'], DS_m]) : \exists x, \exists y,$ such that $(x, \text{op}_t, y) \in DS_m$, $\omega^\alpha(\text{gp}') = x$ and $\omega^\tau(\text{gp}') = y$. Thus, $(x, y) \in \mathcal{L}$ and since $\text{op}_s^\tau = \mathcal{L}$ then $(y, \text{op}_s, x) \in DS_m$, inferring that $\omega \in [[t]]_{DS_m}$.

From the two premises above, we conclude that:

$$[[t]]_{DS_m} \equiv \pi_\mathcal{J}([[\text{gp}'], DS_m])$$

6. Let $\text{op}_t$ be an object property and $c_t$ be a class from the target ontology. Having a mapping $\mu : \text{op}_s \equiv \text{op}_t.\text{domain}(c_t)$ (i.e. $\text{op}_s^\tau = \{(\alpha, b) \mid (\alpha, b) \in \text{op}_t^\tau \land \alpha \in c_t^\tau\}$), the rewritten (based on $t$‘s object part) graph pattern $\text{gp}'$ and its evaluation over the RDF dataset $DS_m$ are the following:

$$\text{gp}' = \mathcal{D}_{\text{gp}}(t, \mu) = (\text{subject}, \text{op}_t, \text{object}) \text{ AND } (\text{subject}, \text{rdf : type}, c_t)$$

$$[[\text{gp}'], DS_m] = [[(\text{subject}, \text{op}_t, \text{object})]_{DS_m} \text{ AND } (\text{subject}, \text{rdf : type}, c_t)]_{DS_m}$$

$$= [[(\text{subject}, \text{op}_t, \text{object})]_{DS_m} \cup [[(\text{subject}, \text{rdf : type}, c_t)]_{DS_m}$$

Let $\mathcal{L} = \{(\alpha, b) \mid (\alpha, b) \in \text{op}_t^\tau \land \alpha \in c_t^\tau\}$. We consider two premises:

(a) $\forall \omega \in [[t]]_{DS_m} : \exists x, \exists y,$ such that $(x, \text{op}_s, y) \in DS_m$, $\omega^\tau(t) = x$ and $\omega^\alpha(t) = y$. Thus, $(x, y) \in \text{op}_s^\tau$ and since $\text{op}_s^\tau = \mathcal{L}$ then $(x, \text{op}_t, y) \in DS_m$ and $(x, \text{rdf : type}, c_t) \in DS_m$, inferring that $\omega \in \pi_\mathcal{J}([[\text{gp}'], DS_m])$.

(b) $\forall \omega \in \pi_\mathcal{J}([[\text{gp}'], DS_m]) : \exists x, \exists y,$ such that $(x, \text{op}_t, y) \in DS_m$, $(x, \text{rdf : type}, c_t) \in DS_m$, $\omega^\alpha(t'_1) = x$ and $\omega^\tau(t'_1) = y$. Thus, $(x, y) \in \mathcal{L}$ and since $\text{op}_s^\tau = \mathcal{L}$ then $(x, \text{op}_s, y) \in DS_m$, inferring that $\omega \in [[t]]_{DS_m}$.

From the two premises above, we conclude that:

$$[[t]]_{DS_m} \equiv \pi_\mathcal{J}([[\text{gp}'], DS_m])$$
7. Let $op_t$ be an object property and $c_t$ be a class from the target ontology. Having a mapping $\mu : op_s \equiv op_t.\text{range}(c_t)$ (i.e. $op^T_s = \{(\alpha, b) \mid (\alpha, b) \in op^T_t \land b \in c^T_t\}$), the rewritten (based on $t$'s object part) graph pattern $gp'$ and its evaluation over the RDF dataset $DS_m$ are the following:

$$gp' = D_{op}(t, \mu) = (\text{subject}, op_t, \text{object}) \text{ AND } (\text{object}, \text{rdf} : \text{type}, c_t)$$

$$[[gp']]_{DS_m} = [[[\text{subject}, op_t, \text{object}]]_{DS_m} \text{ AND } [[[\text{object}, \text{rdf} : \text{type}, c_t]]_{DS_m}$$

$$[[t_1]]_{DS_m} \cup [[[t_2]]_{DS_m}$$

Let $\mathcal{L} = \{(\alpha, b) \mid (\alpha, b) \in op^T_t \land b \in c^T_t\}$. We consider two premises:

(a) $\forall \omega \in [[t]]_{DS_m} : \exists x, \exists y$, such that $(x, op_s, y) \in DS_m$, $\omega^x(t) = x$ and $\omega^y(t) = y$. Thus, $(x, y) \in op^T_s$ and since $op^T_s = \mathcal{L}$ then $(x, op_t, y) \in DS_m$ and $(y, \text{rdf} : \text{type}, c_t) \in DS_m$, inferring that $\omega \in \pi_{\mathcal{J}}([[gp']]_{DS_m})$.

(b) $\forall \omega \in \pi_{\mathcal{J}}([[gp']]_{DS_m}) : \exists x, \exists y$, such that $(x, op_s, y) \in DS_m$, $(y, \text{rdf} : \text{type}, c_t) \in DS_m$, $\omega^x(t'_1) = x$ and $\omega^y(t'_1) = \omega^x(t'_2) = y$. Thus, $(x, y) \in \mathcal{L}$ and since $op^T_t = \mathcal{L}$ then $(x, op_s, y) \in DS_m$, inferring that $\omega \in [[t]]_{DS_m}$.

From the two premises above, we conclude that:

$$[[t]]_{DS_m} \equiv \pi_{\mathcal{J}}([[gp']]_{DS_m})$$

This concludes the proof that the rewriting step of a Data Triple Pattern, based on an object property mapping of its predicate part, is semantics preserving.

Similarly, we prove that the rewriting step performed for a Data Triple Pattern, based on a datatype property mapping of its predicate part, is semantics preserving. Let $dp_s$ be an object property from the source ontology, $t = (\text{subject}, dp_s, \text{object})$ be a Data Triple Pattern and $\mathcal{J}$ be the set of SPARQL variables appearing in $t$. The evaluation of the triple pattern $t$ over the RDF dataset $DS_m$ is presented below:

$$[[t]]_{DS_m} = [[[\text{subject}, dp_s, \text{object}]]_{DS_m}$$

For the different types of datatype property mappings, we consider the following cases:
1. Let $dp_t$ be a datatype property from the target ontology. Having a mapping $\mu : dp_s \equiv dp_t$ (i.e. $dp_s^T = dp_t^T$), the rewritten (based on $t$’s predicate part) graph pattern $gp'$ and its evaluation over the RDF dataset $DS_m$ are the following:

$$gp' = D^p_{dp}(t, \mu) = (subject, dp_t, object)$$

$$[[gp']]_{DS_m} = [[[subject, dp_t, object]]]_{DS_m}$$

We consider two premises:

(a) $\forall \omega \in [t]_{DS_m} : \exists x, \exists y, \text{ such that } (x, dp_s, y) \in DS_m, \, \omega(t) = x \text{ and } \omega^o(t) = y$. Thus, $(x, y) \in dp^T_s$ and since $dp^T_s = dp^T_t$ then $(x, dp_t, y) \in DS_m$, inferring that $\omega \in \pi_J([[gp']]_{DS_m})$.

(b) $\forall \omega \in \pi_J([[gp']]_{DS_m}) : \exists x, \exists y, \text{ such that } (x, dp_t, y) \in DS_m, \, \omega^o(gp') = x \text{ and } \omega^o(gp') = y$. Thus, $(x, y) \in dp^T_t$ and since $dp^T_s = dp^T_t$ then $(x, dp_s, y) \in DS_m$, inferring that $\omega \in [t]_{DS_m}$.

From the two premises above, we conclude that:

$$[t]_{DS_m} \equiv \pi_J([[gp']]_{DS_m})$$

2. Let $op_t$ be an object property and $dp_t$ be a datatype property from the target ontology. Having a mapping $\mu : dp_s \equiv op_t \circ dp_t$ (i.e. $dp_s^T = \{(a, c) \mid \exists b. (a, b) \in op^T_t \land (b, c) \in dp^T_t\}$), the rewritten (based on $t$’s predicate part) graph pattern $gp'$ and its evaluation over the RDF dataset $DS_m$ are the following:

$$gp' = D^p_{dp}(t, \mu) = (subject, op_t, ?var) \text{ AND } (?var, dp_t, object)$$

$$[[gp']]_{DS_m} = [[[subject, op_t, ?var]]]_{DS_m} \text{ AND } [[[?var, dp_t, object]]]_{DS_m}$$

$$= [[[subject, op_t, ?var]]]_{DS_m} \uplus [[[?var, dp_t, object]]]_{DS_m}$$

$$= [[[t_1]]]_{DS_m} \uplus [[[t_2]]]_{DS_m}$$

Let $L = \{(a, c) \mid \exists b. (a, b) \in op^T_t \land (b, c) \in dp^T_t\}$. We consider two premises:

(a) $\forall \omega \in [t]_{DS_m} : \exists x, \exists y, \text{ such that } (x, dp_s, y) \in DS_m, \, \omega(t) = x \text{ and } \omega^o(t) = y$. Thus, $(x, y) \in dp^T_s$ and since $dp^T_s = L$ then $\exists z$ such that $(x, op_t, z) \in DS_m$ and $(z, dp_t, y) \in DS_m$, inferring that $\omega \in \pi_J([[gp']]_{DS_m})$. 


B.3. PROOF OF LEMMA 6.3 139

(b) \( \forall \omega \in \pi_\tau([g^p])_{DS_m} : \exists x, \exists y, \exists z, \) such that \((x, dp_1, z) \in DS_m, (z, dp_2, y) \in DS_m, \omega^s(t'_1) = x, \omega^c(t'_1) = z \) and \(\omega^s(t'_2) = y, \) \((x, y) \in \mathcal{L} \) and since \(dp^s = \mathcal{L} \) then \((x, dp_1, y) \in DS_m, \) inferring that \(\omega \in \{t\}_{DS_m}.\)

From the two premises above, we conclude that:

\[
[t]_{DS_m} = \pi_\tau([g^p])_{DS_m}
\]

3. Let \(dp_{t1}\) and \(dp_{t2}\) be datatype properties from the target ontology. Having a mapping \(\mu : dp_s \equiv \{dp_{t1} \cup dp_{t2}\} \) (i.e. \(dp^s = dp^T_{t1} \cup dp^T_{t2}\)), the rewritten (based on \(t\)'s predicate part) graph pattern \(g^p\) and its evaluation over the RDF dataset \(DS_m\) are the following:

\[
g^p' = \mathcal{D}_{dp}(t, \mu) = (subject, dp_{t1}, object) \cup (subject, dp_{t2}, object)
\]

\[
[[g^p']]_{DS_m} = \{[\{subject, dp_{t1}, object\}]_{DS_m} \cup \{[subject, dp_{t2}, object]\}]_{DS_m}
\]

We consider two premises:

(a) \( \forall \omega \in \{t\}_{DS_m} : \exists x, \exists y, \) such that \((x, dp_s, y) \in DS_m, \omega^s(t) = x \) and \(\omega^c(t) = y.\)

Thus, \((x, y) \in dp^s \) and since \(dp^s = dp^T_{t1} \cup dp^T_{t2}\) then \((x, dp_{t1}, y) \in DS_m \) or \((x, dp_{t2}, y) \in DS_m, \) inferring that \(\omega \in \pi_\tau([g^p')]_{DS_m}).\)

(b) \( \forall \omega \in \pi_\tau([g^p'])_{DS_m} : \exists x, \exists y, \) such that \((x, dp_{t1}, y) \in DS_m \) or \((x, dp_{t2}, y) \in DS_m, \omega^s(t'_1) = x \) or \(\omega^c(t'_1) = x, \) \(\omega^s(t'_2) = y \) or \(\omega^c(t'_2) = y.\)

Thus, \((x, y) \in dp^T_{t1} \cup dp^T_{t2}\) and since \(dp^s = dp^T_{t1} \cup dp^T_{t2}\) then \((x, dp_s, y) \in DS_m, \) inferring that \(\omega \in \{t\}_{DS_m} \).

From the two premises above, we conclude that:

\[
[t]_{DS_m} = \pi_\tau([g^p'])_{DS_m}
\]

4. Let \(dp_{t1}\) and \(dp_{t2}\) be datatype properties from the target ontology. Having a mapping \(\mu : dp_s \equiv dp_{t1} \cap dp_{t2}\) (i.e. \(dp^s = dp^T_{t1} \cap dp^T_{t2}\)), the rewritten (based on \(t\)'s predicate part) graph pattern \(g^p\) and its evaluation over the RDF dataset \(DS_m\) are the following:
\[ gp' = D^\mu_{dp'}(t, \mu) = (\text{subject}, dp_{t1}, \text{object}) \text{ AND } (\text{subject}, dp_{t2}, \text{object}) \]

\[ [[gp']]_{DS_m} = [[[\text{subject}, dp_{t1}, \text{object}) \text{ AND } (\text{subject}, dp_{t2}, \text{object})]]]_{DS_m} \]

We consider two premises:

(a) \( \forall \omega \in [[t]]_{DS_m} : \exists x, \exists y, \text{such that } (x, dp_x, y) \in DS_m, \omega^x(t) = x \text{ and } \omega^y(t) = y. \)
Thus, \( (x, y) \in dp_x^T \) and since \( dp_x^T = dp_{t1}^T \land dp_{t2}^T \) then \( (x, dp_{t1}, y) \in DS_m \) and \( (x, dp_{t2}, y) \in DS_m \), inferring that \( \omega \in \pi_\omega([[gp']]_{DS_m}). \)

(b) \( \forall \omega \in \pi_\omega ([[gp']]_{DS_m}) : \exists x, \exists y, \text{such that } (x, dp_{t1}, y) \in DS_m, (x, dp_{t2}, y) \in DS_m, \omega^x(t_1') = \omega^x(t_2') = x \text{ and } \omega^y(t_1') = \omega^y(t_2') = y. \)
Thus, \( (x, y) \in dp_{t1}^T \land dp_{t2}^T \) and since \( dp_x^T = dp_{t1}^T \land dp_{t2}^T \) then \( (x, dp_x, y) \in DS_m \), inferring that \( \omega \in [[t]]_{DS_m}. \)

From the two premises above, we conclude that:

\[ [[t]]_{DS_m} \equiv \pi_\omega([[gp']]_{DS_m}) \]

5. Let \( dp_t \) be a datatype property and \( c_t \) be a class from the target ontology. Having a mapping \( \mu : dp_t \equiv dp_t, \text{domain}(c_t) \) (i.e. \( dp_t^T = \{(a, b) | (a, b) \in dp_t^T \land \alpha \in c_t^T\} \)), the rewritten (based on \( t \)'s object part) graph pattern \( gp' \) and its evaluation over the RDF dataset \( DS_m \) are the following:

\[ gp' = D^\mu_{dp'}(t, \mu) = (\text{subject}, dp_t, \text{object}) \text{ AND } (\text{subject}, rdf : type, c_t) \]

\[ [[gp']]_{DS_m} = [[[\text{subject}, dp_t, \text{object}) \text{ AND } (\text{subject}, rdf : type, c_t)]]]_{DS_m} \]

Let \( \mathcal{L} = \{(a, b) | (a, b) \in dp_t^T \land \alpha \in c_t^T\}. \) We consider two premises:

(a) \( \forall \omega \in [[t]]_{DS_m} : \exists x, \exists y, \text{such that } (x, dp_x, y) \in DS_m, \omega^x(t) = x \text{ and } \omega^y(t) = y. \)
Thus, \( (x, y) \in dp_x^T \) and since \( dp_x^T = \mathcal{L} \) then \( (x, dp_t, y) \in DS_m \) and \( (x, rdf : type, c_t) \in DS_m \), inferring that \( \omega \in \pi_\omega([[gp']]_{DS_m}). \)
(b) \( \forall \omega \in \pi_\gamma (\langle [gp'] DS_m \rangle) : \exists x, \exists y, \text{ such that } (x, dp_s, y) \in DS_m, (x, rdf : type, c_t) \in DS_m, \omega^\gamma (t'_1) = \omega^\gamma (t'_2) = x \text{ and } \omega^\gamma (t'_1) = y. \) Thus, \( (x, y) \in \mathcal{L} \) and since \( dp_s^T = \mathcal{L} \) then \( (x, dp_s, y) \in DS_m, \) inferring that \( \omega \in \langle [t] DS_m \rangle. \) From the two premises above, we conclude that:

\[
\langle [t] DS_m \rangle \equiv \pi_j (\langle [gp'] DS_m \rangle)
\]

6. Let \( dp_t \) be a datatype property from the target ontology, \( v_{dp} \) be a data value and \( cp \in \{ \neq, =, >, <, \geq, \leq \}. \) Having a mapping \( \mu : dp_s \equiv dp_t. range (cp v_{dp}) \) (i.e. \( dp_s^T = \{(\alpha, b) \mid (\alpha, b) \in dp_t^T \land b \ cp v_{dp}\} \)), the rewritten (based on \( t \)'s object part) graph pattern \( gp' \) and its evaluation over the RDF dataset \( DS_m \) are the following:

\[
gp' = D_{dp_t}(t, \mu) = (subject, dp_t, object) \ \text{FILTER}(object \ cp v_{dp})
\]

\[
\langle [gp'] DS_m \rangle = \langle (subject, dp_t, object) \ \text{FILTER}(object \ cp v_{dp}) \rangle DS_m = \{\omega \in \langle (subject, dp_t, object) \rangle DS_m \mid \omega \models object \ cp v_{dp} \}
\]

\[
= \{\omega \in \langle [t'] DS_m \rangle \mid \omega \models \text{object } cp v_{dp} \}
\]

Let \( \mathcal{L} = \{(\alpha, b) \mid (\alpha, b) \in dp_t^T \land b \ cp v_{dp}\}. \) We consider two premises:

(a) \( \forall \omega \in \langle [t] DS_m \rangle : \exists x, \exists y, \text{ such that } (x, dp_s, y) \in DS_m, \omega^\gamma (t) = x \text{ and } \omega^\gamma (t) = y. \) Thus, \( (x, y) \in dp_s^T \) and since \( dp_s^T = \mathcal{L} \) then \( (x, dp_s, y) \in DS_m \) and \( y \ cp v_{dp}, \) inferring that \( \omega \in \pi_\gamma (\langle [gp'] DS_m \rangle). \)

(b) \( \forall \omega \in \pi_\gamma (\langle [gp'] DS_m \rangle) : \exists x, \exists y, \text{ such that } (x, dp_s, y) \in DS_m, y \ cp v_{dp}, \omega^\gamma (t') = x \text{ and } \omega^\gamma (t') = y. \) Thus, \( (x, y) \in \mathcal{L} \) and since \( dp_s^T = \mathcal{L} \) then \( (x, dp_s, y) \in DS_m, \) inferring that \( \omega \in \langle [t] DS_m \rangle. \)

From the two premises above, we conclude that:

\[
\langle [t] DS_m \rangle \equiv \pi_j (\langle [gp'] DS_m \rangle)
\]

This concludes the proof that the rewriting step of a Data Triple Pattern, based on a datatype property mapping of its predicate part, is semantics preserving.
Appendix C

Mapping representation ontology

In this appendix, we provide the OWL DL ontology which describes the mapping language’s vocabulary and provides some control to the terms and constructs introduced in Section 4.4.1. As we mentioned in Section 4.4, the mapping representation language that we adapt in this thesis is a new version of EDOAL. In this version minor changes have been performed in the EDOAL syntax, in order for the language to match exactly the abstract syntax presented in Section 4.2 and also to restrict the language expressiveness for simplicity reasons. EDOAL previous versions have been defined in [19] and [35]. The expressiveness, the simplicity and the Semantic Web compliance (given its RDF syntax) are its key features. It is crucial that this language satisfies the requirements that we set in Section 4.3, by representing clearly the supported mapping types.

```xml
<?xml version="1.0"?>

<!DOCTYPE rdf:RDF [
  <!ENTITY owl "http://www.w3.org/2002/07/owl#" >
  <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
  <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
  <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
  <!ENTITY align "http://knowledgeweb.semanticweb.org/heterogeneity/alignment#" >
]

<rdf:RDF xmlns="http://www.music.tuc.gr/oml#"
  xml:base="http://www.music.tuc.gr/oml#"
  xmlns:align="http://knowledgeweb.semanticweb.org/heterogeneity/alignment#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:owl="http://www.w3.org/2002/07/owl#">
  <owl:Ontology rdf:about="#"/>
</rdf:RDF>
```
<owl:Class rdf:about="#align:Ontology">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty>
        <owl:ObjectProperty rdf:about="#align:formalism"/>
      </owl:onProperty>
      <owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
    </owl:Restriction>
    <rdfs:subClassOf rdf:resource="#owl:Thing"/>
  </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:ID="#PropertyDomainRestriction">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty>
        <owl:ObjectProperty rdf:about="#class"/>
      </owl:onProperty>
      <owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
    </owl:Restriction>
    <rdfs:subClassOf rdf:resource="#Restriction"/>
  </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:ID="#PropertyValueRestriction">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty>
        <owl:DatatypeProperty rdf:about="#literalValue"/>
      </owl:onProperty>
      <owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
    </owl:Restriction>
    <rdfs:subClassOf rdf:resource="#Restriction"/>
  </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:ID="#RelationRangeRestriction">
  <rdfs:subClassOf rdf:resource="#Restriction"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
      <owl:onProperty>
        <owl:ObjectProperty rdf:about="#class"/>
      </owl:onProperty>
    </owl:Restriction>
    <rdfs:subClassOf rdf:resource="#Restriction"/>
  </rdfs:subClassOf>
</owl:Class>
<owl:onProperty>
  <owl:DatatypeProperty rdf:about="#comparator"/>
</owl:onProperty>
<owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
</owl:Restriction>
</rdfs:subClassOf>
</owl:Class>
<br:Class rdf:about="&align;Alignment">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
    </owl:Restriction>
    <owl:Restriction>
      <owl:onProperty>
        <owl:ObjectProperty rdf:about="&align;onto1"/>
      </owl:onProperty>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
    </owl:Restriction>
    <owl:Restriction>
      <owl:onProperty>
        <owl:DatatypeProperty rdf:about="&align;method"/>
      </owl:onProperty>
    </owl:Restriction>
    <owl:maxCardinality rdf:datatype="&xsd;int">1</owl:maxCardinality>
  </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
    </owl:Restriction>
    <owl:Restriction>
      <owl:onProperty>
        <owl:DatatypeProperty rdf:about="&align;type"/>
      </owl:onProperty>
    </owl:Restriction>
    <owl:maxCardinality rdf:datatype="&xsd;int">1</owl:maxCardinality>
  </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
    </owl:Restriction>
    <owl:Restriction>
      <owl:onProperty>
        <owl:DatatypeProperty rdf:about="&align;level"/>
      </owl:onProperty>
    </owl:Restriction>
    <owl:maxCardinality rdf:datatype="&xsd;int">1</owl:maxCardinality>
  </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:minCardinality rdf:datatype="&xsd;int">1</owl:minCardinality>
    </owl:Restriction>
    <owl:Restriction>
      <owl:ObjectProperty rdf:about="&align;map"/>
    </owl:Restriction>
  </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf rdf:resource="&owl;Thing"/>
  </rdfs:subClassOf>
<owl:Restriction>
  <owl:maxCardinality rdf:datatype="&xsd;int">1</owl:maxCardinality>
  <owl:onProperty>
    <owl:DatatypeProperty rdf:about="&align;xml"/>
  </owl:onProperty>
</owl:Restriction>
</rdfs:subClassOf>
<rdfs:subClassOf>
  <owl:Restriction>
    <owl:onProperty>
      <owl:DatatypeProperty rdf:about="&align;xml"/>
    </owl:onProperty>
    <owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
  </owl:Restriction>
</rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:ID="Class">
  <rdfs:subClassOf>
    <owl:Class rdf:ID="Entity"/>
  </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:about="#Attribute">
  <rdfs:subClassOf>
    <owl:Class rdf:about="#Entity"/>
  </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:about="&align;Cell">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
      <owl:onProperty>
        <owl:ObjectProperty rdf:about="&align;entity2"/>
      </owl:onProperty>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty>
        <owl:ObjectProperty rdf:about="&align;entity1"/>
      </owl:onProperty>
      <owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty>
        <owl:DatatypeProperty rdf:about="&align;relation"/>
      </owl:onProperty>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
</rdfs:subClassOf>
  <rdfs:subClassOf rdf:resource="&owl;Thing"/>
</owl:Class>
<owl:Class rdf:about="&align;Formalism">
  <rdfs:subClassOf rdf:resource="&owl;Thing"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:minCardinality rdf:datatype="&xsd;int">1</owl:minCardinality>
      <owl:onProperty>
        <owl:DatatypeProperty rdf:about="&align;name"/>
      </owl:onProperty>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:about="#Entity">
  <rdfs:subClassOf rdf:resource="&owl;Thing"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
      <owl:onProperty>
        <owl:DatatypeProperty rdf:about="&align;uri"/>
      </owl:onProperty>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:ID="Property">
  <rdfs:subClassOf rdf:resource="#Attribute"/>
</owl:Class>
<owl:Class rdf:ID="Instance">
  <rdfs:subClassOf rdf:resource="#Entity"/>
</owl:Class>
<owl:ObjectProperty rdf:ID="inverse">
  <rdfs:subPropertyOf>
    <owl:ObjectProperty rdf:about="#constructor"/>
  </rdfs:subPropertyOf>
  <rdfs:domain rdf:resource="#Relation"/>
  <rdfs:range rdf:resource="#Relation"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="&align;onto2">
  <rdfs:subPropertyOf>
    <owl:ObjectProperty rdf:about="&align;ontology"/>
  </rdfs:subPropertyOf>
</owl:ObjectProperty>
<owl:ObjectProperty>
<owl:ObjectProperty rdf:about="&align;map">
  <rdfs:domain rdf:resource="&align;Alignment"/>
  <rdfs:range rdf:resource="&align;Cell"/>  
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="&align;entity2">
  <owl:ObjectProperty rdf:about="&align;entity"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="&align;ontology">
  <rdfs:range rdf:resource="&align;Ontology"/>
  <rdfs:domain rdf:resource="&align;Alignment"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="restrict">
  <rdfs:domain>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Class"/>
        <owl:Class rdf:about="#Property"/>
        <owl:Class rdf:about="#Relation"/>
      </owl:unionOf>
      <owl:Class rdf:about="#Relation"/>
    </owl:Class>
  </rdfs:domain>
  <rdfs:subPropertyOf>
    <owl:ObjectProperty rdf:about="#constructor"/>
  </rdfs:subPropertyOf>
  <rdfs:range>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Entity"/>
        <owl:Class rdf:about="#Restriction"/>
      </owl:unionOf>
      <owl:Class rdf:about="#Restriction"/>
    </owl:Class>
  </rdfs:range>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="&align;formalism">
  <rdfs:range rdf:resource="&align;Formalism"/>
  <rdfs:domain rdf:resource="&align;Ontology"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#onAttribute">
  <rdfs:domain rdf:resource="#AttributeValueRestriction"/>
  <rdfs:range rdf:resource="#Attribute"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#instanceOrAttributeValue">
  <rdfs:domain rdf:resource="#AttributeValueRestriction"/>
  <rdfs:range rdf:resource="#Instance"/>
</owl:ObjectProperty>
<owl:Class rdf:about="#Relation"/>
<owl:Class rdf:about="#Property"/>
</owl:unionOf>
</owl:Class>
</rdfs:range>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#constructor">
  <rdfs:domain>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Class"/>
        <owl:Class rdf:about="#Property"/>
        <owl:Class rdf:about="#Relation"/>
      </owl:unionOf>
    </owl:Class>
  </rdfs:domain>
</owl:ObjectProperty>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="and">
  <rdfs:range>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Class"/>
        <owl:Class rdf:about="#Property"/>
        <owl:Class rdf:about="#Relation"/>
      </owl:unionOf>
    </owl:Class>
  </rdfs:range>
  <rdfs:domain>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Class"/>
        <owl:Class rdf:about="#Property"/>
        <owl:Class rdf:about="#Relation"/>
      </owl:unionOf>
    </owl:Class>
  </rdfs:domain>
  <rdfs:subPropertyOf rdf:resource="#constructor"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#align:entity">
  <rdfs:domain rdf:resource="#align:Cell"/>
  <rdfs:range rdf:resource="#Entity"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="compose">
  <rdfs:range>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Property"/>
        <owl:Class rdf:about="#Relation"/>
      </owl:unionOf>
    </owl:Class>
  </rdfs:range>
</owl:ObjectProperty>
<owl:DatatypeProperty rdf:about="&align;name">
  <rdfs:domain rdf:resource="&align;Formalism"/>
  <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="&align;type">
  <rdfs:domain rdf:resource="&align;Alignment"/>
  <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="&align;method">
  <rdfs:domain rdf:resource="&align;Alignment"/>
  <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="#literalValue">
  <rdfs:domain>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#PropertyValueRestriction"/>
        <owl:Class rdf:about="#AttributeValueRestriction"/>
      </owl:unionOf>
    </owl:Class>
  </rdfs:domain>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="&align;level">
  <rdfs:domain rdf:resource="&align;Alignment"/> 
  <rdfs:range rdf:resource="&xsd;int"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="&align;relation">
  <rdfs:range>
    <owl:DataRange>
      <owl:oneOf rdf:parseType="Resource">
        <rdf:first rdf:datatype="&xsd;string">Equivalence</rdf:first>
        <rdf:rest rdf:parseType="Resource">
          <rdf:first rdf:datatype="&xsd;string">Subsumes</rdf:first>
          <rdf:rest rdf:parseType="Resource">
            <rdf:first rdf:datatype="&xsd;string">SubsumedBy</rdf:first>
            <rdf:rest rdf:resource="&rdf;nil"/>
          </rdf:rest>
        </rdf:rest>
      </owl:oneOf>
    </owl:DataRange>
  </rdfs:range>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="&align;xml">
  <rdfs:range>
    <owl:DataRange>
      <owl:oneOf rdf:parseType="Resource">
        <rdf:first rdf:datatype="&xsd;string">yes</rdf:first>
        <rdf:rest rdf:parseType="Resource">
          <rdf:first rdf:datatype="&xsd;string">no</rdf:first>
        </rdf:rest>
      </owl:oneOf>
    </owl:DataRange>
  </rdfs:range>
</owl:DatatypeProperty>
Appendix D

Mapping representation example

In this appendix, we provide the representation of the mappings defined in Tables 4.8, 4.9, 4.10 and 4.11, for the ontologies presented in Figure 4.1. For the mapping representation, we use the language discussed in the Section 4.4 and the ontology presented in the Appendix C.

<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [
   <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#">
   <!ENTITY src "http://www.ontologies.com/SourceOntology.owl#">
   <!ENTITY trg "http://www.ontologies.com/TargetOntology.owl#">
]>

<rdf:RDF xmlns="http://www.alignments.com/example.owl#"
         xml:base="http://www.alignments.com/example.owl#"
         xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
         xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
         xmlns:src="http://www.ontologies.com/SourceOntology.owl#"
         xmlns:trg="http://www.ontologies.com/TargetOntology.owl#"
         xmlns:align="http://knowledgeweb.semanticweb.org/heterogeneity/alignment#"
         xmlns:oml="http://www.music.tuc.gr/oml#">
   <align:Alignment rdf:about="http://www.alignments.com/example.owl#">
      <align:Ontology rdf:about="&src;">
         <align:formalism>
            <align:Formalism>
               <align:uri>http://www.w3.org/TR/owl-guide/</align:uri>
               <align:name>OWL</align:name>
            </align:Formalism>
         </align:formalism>
      </align:Ontology>
      <align:Ontology rdf:about="&trg;">
         <align:formalism>
            <align:Formalism>
               <align:uri>http://www.w3.org/2000/01/rdf-schema#</align:uri>
               <align:name>RDFS</align:name>
            </align:Formalism>
         </align:formalism>
      </align:Ontology>
   </align:Alignment>
</rdf:RDF>
<align:map>
  <align:Cell rdf:about="MappingRule_i">
    <align:entity1>
      <oml:Class rdf:about="&src;NewPublication"/>
    </align:entity1>
    <align:entity2>
      <oml:Class>
        <oml:and rdf:parseType="Collection">
          <oml:Class rdf:about="&trg;Computing"/>
          <oml:Class rdf:about="&trg;NewRelease"/>
        </oml:and>
      </oml:Class>
    </align:entity2>
    <align:relation>Equivalence</align:relation>
  </align:Cell>
</align:map>
<align:map>
  <align:Cell rdf:about="MappingRule_j">
    <align:entity1>
      <oml:Class rdf:about="&src;Science"/>
    </align:entity1>
    <align:entity2>
      <oml:Class>
        <oml:or rdf:parseType="Collection">
          <oml:Class rdf:about="&trg;ComputerScience"/>
          <oml:Class rdf:about="&trg;Mathematics"/>
        </oml:or>
      </oml:Class>
    </align:entity2>
    <align:relation>Equivalence</align:relation>
  </align:Cell>
</align:map>
<align:map>
  <align:Cell rdf:about="MappingRule_k">
    <align:entity1>
      <oml:Class rdf:about="&src;Popular"/>
    </align:entity1>
  </align:Cell>
</align:map>
<align:map>
  <align:Cell rdf:about="MappingRule_t">
    <align:entity1>
      <oml:Instance rdf:about="&src;CSFoundations"/>
    </align:entity1>
    <align:entity2>
      <oml:Instance rdf:about="&trg;FoundationsOfCS"/>
    </align:entity2>
    <align:relation>Equivalence</align:relation>
  </align:Cell>
  </align:map>
</align:Alignment>
</rdf:RDF>