THE SPARQL-RW FRAMEWORK

MAPPING MODELING AND QUERY REWRITING FOR ONTOLOGY BASED MEDIATORS

By

Konstantinos E. Makris

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE
IN ELECTRONIC & COMPUTER ENGINEERING

at the

SCHOOL OF ELECTRONIC & COMPUTER ENGINEERING
TECHNICAL UNIVERSITY OF CRETE

2014

I hereby declare that I am the sole author of this thesis.

I authorize the Technical University of Crete to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the Technical University of Crete to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Konstantinos E. Makris
Technical University of Crete
May 2014

Abstract

The Web of Data is an open environment consisting of heterogeneous, distributed and highly structured information sources. Uniform information access in this kind of setting is of major importance for data consumer applications and end users. To this end, ontology based mediator systems supporting transparent query access over federated data sources are considered essential. In this thesis we present SPARQL–RW, a Framework supporting mapping modeling and query rewriting in the context of ontology based mediator architectures. SPARQL–RW provides a formal model for describing mappings between ontology schemas, as well as a generic method for SPARQL 1.1 query rewriting, with respect to a set of predefined ontology mappings and data source endpoints. The mapping model supports a set of rich and flexible mapping types based on Description Logic semantics, as well as notable mapping formalisms, including *Global-As-View (GAV)*, *Local-As-View (LAV)*, and *Global-and-Local-As-View (GLAV)*. Additionally, it defines a mapping language capable of representing all the supported types of inter-schema correspondences. The Framework provides functionality for performing mapping inference and for identifying inconsistencies in a given set of mappings and ontology schemas. Regarding query rewriting, the proposed algorithms are proved to provide semantics preserving queries with respect to the GAV mapping types supported by the model. The reformulated queries can be executed directly on any SPARQL federated query engine, or exploited as logical query plans by any ontology based mediator system. SPARQL–RW has been implemented, formally evaluated and tested in a prototype mediator system integrating several data providers from the biodiversity community along with DBpedia.

Acknowledgements

I would like to express my gratitude to my supervisor, Prof. Stavros Christodoulakis, for his encouragement and his continuous guidance and support throughout my research. I would also like to thank him for the important experiences he offered me during my stay at the Laboratory of Distributed Multimedia Information Systems and Applications (TUC/MUSIC), as well as for the positive influence in broadening my horizons.

My sincerest thanks also go to Nektarios Gioldasis and Giannis Skevakis for their continuous support, their valuable advices, as well as our long fruitful discussions not only regarding this thesis but also throughout my stay at the TUC/MUSIC Laboratory. I would also like to thank Prof. Antonios Deligiannakis and Prof. Michail G. Lagoudakis for serving on my thesis committee.

My appreciation goes to Polyxeni Arapi, Fotis Kazasis and Nikos Bikakis for being always ready to offer their help whenever needed. Furthermore, I would like to thank all my colleagues in the TUC/MUSIC Laboratory for their support and for the pleasant environment they provided.

Finally, this thesis would not have been possible without the support and encouragement of five special people: my father Manolis, my mother Kiriaki, my brothers Zisis and Dimitris, and my other half Helen. Thank you from the bottom of my heart for your unconditional love and care.

Bibliographical Notes

Preliminary versions of the material presented in this thesis, as well as parts of the overall work carried out during my Master's studies have been published in the following journals and conference proceedings:

- Makris K., Bikakis N., Gioldasis N., Christodoulakis S.: "SPARQL-RW: Transparent Query Access over Mapped RDF Data Sources". In Proceedings of the 15th International Conference on Extending Database Technology (EDBT), 2012.
- Makris K., Gioldasis N., Bikakis N., Christodoulakis S.: "Ontology Mapping and SPARQL Rewriting for Querying Federated RDF Data Sources". In Proceedings of the 9th International Conference on Ontologies, DataBases, and Applications of Semantics (ODBASE), 2010.
- Makris K., Bikakis N., Gioldasis N., Tsinaraki C., Christodoulakis S.: "Towards a Mediator based on OWL and SPARQL". In Proceedings of the 2nd World Summit on the Knowledge Society (WSKS), 2009.

- Skevakis G., Makris K., Kalokyri V., Arapi P., Christodoulakis S.: "Metadata Management, Interoperability and Linked Data Publishing Support for Natural History Museums". International Journal on Digital Libraries (IJDL), 2014.
- ☆ Makris K., Skevakis G., Kalokyri V., Arapi P., Christodoulakis S., Stoitsis J., Manolis N., Rojas S. L.: "Federating Natural History Museums in Natural Europe". In Proceedings of the 7th Metadata and Semantics Research Conference (MTSR), 2013. *Best Student Paper Award*.
- ☆ Makris K., Skevakis G., Kalokyri V., Arapi P., Christodoulakis S.: "Metadata Management and Interoperability Support for Natural History Museums". In Proceedings of the 17th International Conference on Theory and Practice of Digital Libraries (TPDL), 2013. *Best Student Paper Nomination*.
- Skevakis G., Makris K., Arapi P., Christodoulakis S.: "Elevating Natural History Museums' Cultural Collections to the Linked Data Cloud". In Proceedings of the 3rd International Workshop on Semantic Digital Archives (SDA), 2013.
- Makris K., Skevakis G., Kalokyri V., Gioldasis N., Kazasis F., Christodoulakis S.: "Bringing Environmental Culture Content into the Europeana.eu Portal: The Natural Europe Digital Libraries Federation Infrastructure". In Proceedings of the 5th Metadata and Semantics Research Conference (MTSR), 2011.

I would like to thank my co-authors for their assistance and the anonymous reviewers for their helpful feedback on the issues discussed in these papers and articles.

Dedicated to my family

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The Web of Data is an environment that allows publishing data on the Web, in structured, linked, and standardized ways. It is comprised by a large number of inter-linked RDF datasets from different domains, and initiatives like Linked Open Data, Open Government and Linked Life Data have played a major role towards its development.

In this environment, it is common for independent institutions and organizations to expose data of the same or overlapping domain using different conceptualizations and not a globally shared ontological scheme. A plethora of such examples can be given, starting from the *DBpedia* [9], *YAGO* [86] and *Freebase* [10] cross-domain datasets, *ACM*, *IEEE*, *DBLP* and *ePrints* in the domain of publications, *PubMed*, *GeneID*, *Drug Bank* and *Gen Bank* in life science, as well as *GeoNames*, *Linked GeoData* and *Geo Linked Data* in the geographic domain. Numerous other examples can be obtained from the Web of Data graph. Attempts to find agreements for common conceptualizations often result in semantically weak minimum consensus schemes, like *Dublin Core* [48], or models with extensive and complex semantics, like *CIDOC/CRM* [22]. Moreover, in many cases cooperating institutions and organizations have their own proprietary conceptualizations and it is not often feasible for them to agree on a certain model or apply an existing standard.

Considering that information retrieval from RDF data sources needs to take into account the data semantics at the conceptual level, it becomes obvious that systems supporting transparent querying over distributed and federated datasets are essential components for a great number of Web of Data applications. Although many state of the art systems, like *LDIF* [79], *SPARQL++* [69], and *Mosto* [75], are focused on the RDF data exchange/transformation problem, to the best of our knowledge, there is no system supporting transparent querying over multiple mapped RDF data sources. Such systems are usually employed by mediator based architectures [91], offering a single point for querying access to the integrated data sources and requiring from the end-users to be only aware of the mediator schema.

In this thesis, we present the SPARQL–RW Framework. SPARQL–RW provides a formal model

for describing mappings between OWL ontology schemas, as well as a generic method for SPARQL 1.1 query rewriting, with respect to a set of predefined ontology mappings and data source endpoints. The mapping model supports a set of rich and flexible mapping types based on Description Logic (DL) [5] semantics, as well as notable mapping formalisms, including: (*a*) *Global-As-View (GAV)* [29], which is based on the idea that the mediator schema is described as a set of views over the source schemas, (*b*) *Local-As-View (LAV)* [52], in which the source schemas are described as views over the mediator schema, and (*c*) *Global-and-Local-As-View (GLAV)* [54], which combines both GAV and LAV. Additionally, the mapping model defines a language capable of representing all the supported types of inter-schema correspondences. The Framework provides functionality for performing mapping inference and for identifying inconsistencies in a given set of mappings and ontology schemas. Regarding query rewriting, the proposed algorithms are proved to provide semantics preserving queries with respect to the GAV mapping types supported by the model. The reformulated queries can be executed directly on any SPARQL federated query engine, or exploited as logical query plans by any ontology based mediator system.

Formally, let G be a global ontology schema, let $\bar{S} = \{S_1, \ldots, S_n\}$ be n data source ontology schemas, and let $\bar{M}$ be a set of mappings between G and $\bar{S}$. SPARQL–RW takes as input a SPARQL query Q_G expressed over G, and rewrites it to a semantically correspondent query $Q_{\bar{S}}$, expressed over $\bar{S}$, with respect to $\bar{M}$. Subsequently, considering a set of endpoints $\bar{E} = \{E_1, \ldots, E_m\}$, and a set of relations <$\bar{S}, \bar{E}$>, specifying the available endpoints for each integrated ontology schema, SPARQL–RW transforms the reformulated query $Q_{\bar{S}}$ to a federated one, expressed over $\bar{E}$.

In more detail, the aim and the objectives of this thesis are to: (*a*) *determine the types of ontology mappings that can be exploited in the SPARQL query rewriting process*, (*b*) *specify a mapping model* supporting well-known mapping formalisms and providing the basic constructs to describe ontology mappings of certain types, (*c*) *investigate inferencing of new ontology mappings* and *study the identification of mapping inconsistencies* in a given set of mappings and ontology schemas, (*d*) *define a method for the rewriting of SPARQL 1.1 queries* based on the mapping types and formalisms supported by the specified mapping model, and finally (*e*) *evaluate the query rewriting method* in terms of soundness, completeness and efficiency.

1.1 Summary of Contributions

In this thesis we present a Framework for supporting mapping modeling and query rewriting in the context of ontology based mediator architectures.

In more detail, we propose a model for the expression of mappings between ontology schemas. The mapping model consists of a grammar describing the mapping types which can be exploited in SPARQL query rewriting, as well as a specification of the mapping type semantics. It is based on Description Logics and it is able to support a great variety of mappings between OWL ontologies,

providing high flexibility and satisfying different system requirements and user needs. Furthermore, it is able to support a variety of mapping formalisms, including *GAV*, *LAV*, and *GLAV*, satisfying strong data integration requirements for *query rewriting efficiency* and *extensibility to new sources*. To the best of our knowledge, there is no system supporting these formalisms in the field of ontology based mediators.

Additionally to the mapping model, we define a language based on XML syntax, being able to represent the discussed mapping types and formalisms. The mapping language combines a set of criteria including *simplicity*, *expressiveness*, *executability*, and *schema language agnosticism*. Furthermore, the use of XML Schema in mapping language definition: (*a*) provides exceptional *validation* capabilities, (*b*) supports easy mapping *serialization and deserialization*, and (*c*) enables *interoperability* with external systems and applications.

We support the maintenance of mappings conforming to the SPARQL–RW mapping model by providing a method for identifying mapping inconsistencies in a given set of mappings and ontology schemas. Moreover, in order to assist the mapping definition process we exploit the DL semantics of the SPARQL–RW mapping model for performing mapping inference through the use of well-known reasoning techniques.

Regarding query rewriting, we provide a formal method for the reformulation of SPARQL 1.1 queries posed over the mediator, into federated SPARQL queries referring to the integrated data sources. The query rewriting process is mainly *based on a complete set of inference rules and recursive transformation functions, as well as on the exploitation of ontology mappings described using the SPARQL–RW mapping model*. The provided algorithms and transformation functions are formally evaluated for their soundness and completeness, and are *proved to provide semantics preserving queries* with respect to the GAV inter-schema correspondences supported by the model. To the extent of our knowledge, there is no system performing SPARQL 1.1 query rewriting in general, or SPARQL query rewriting by exploiting well-known mapping formalisms in the context of ontology based mediators.

Finally, the SPARQL–RW Framework has been fully implemented and evaluated in terms of its query rewriting efficiency, measuring the time required for the reformulation of queries of different size and type, using mappings of varying complexity. Furthermore, it has been tested in the Semantic Query Mediation Prototype Infrastructure that we have developed, supporting the integration of DBpedia [9] and several biodiversity data providers from the Natural Europe project [1]. This infrastructure enables the execution of highly sophisticated queries combining specimen and media object information persisted in the Natural Europe repositories with species information available in DBpedia. External applications and end-users are able to interact with a single ontology schema and endpoint, without having to be aware of the schemas and SPARQL endpoints of the integrated data sources. As a result, highly complex queries, combining information from multiple data sources, can be expressed in a few triples, reducing drastically the effort of query composition.

1.2 Reader's Guide

Apart from the introduction, the preliminaries, the related work, and the conclusion, this thesis can be divided into three parts. In the first part, we define the mapping model and language, as well as investigate mapping inference and inconsistency identification. In the second part, we describe our query rewriting method, by providing certain algorithms and transformation functions, considering input queries executed either on data or schema. Finally, the third part discusses the implementation of the specified mapping model and query rewriting algorithms, as well as the evaluation and testing of the proposed Framework. More precisely, this thesis is structured as follows:

- Chapter 2 presents the most relevant research to the issues addressed in this thesis. It describes their approach and discusses their advantages or disadvantages compared to ours.
- Chapter 3 provides a brief overview of the RDF data model and OWL, the standard language for defining and instantiating Web ontologies. Moreover, it introduces SPARQL focusing on the language syntax and semantics.
- Chapter 4 specifies the abstract syntax and semantics of the SPARQL–RW mapping model and discusses the supported mapping formalisms. In addition, it provides an analysis of the mapping model expressiveness by comparing its features to well-known knowledge and mapping representation languages. Finally, it describes methods for performing mapping inference and inconsistency identification, and defines a mapping representation language by providing its general structure along with illustrative mapping examples.
- Chapter 5 presents the developed algorithms for performing SPARQL 1.1 query rewriting using GAV ontology mappings supported by the SPARQL–RW mapping model. The provided algorithms are based on a complete set of inference rules and recursive DL to SPARQL transformation functions and consider both data and schema queries.
- Chapter 6 discusses the experimental evaluation conducted on SPARQL–RW in terms of its query rewriting efficiency and provides the obtained results. Moreover, it presents the Semantic Query Mediation Prototype Infrastructure that we have developed in order to demonstrate the applicability of the SPARQL–RW Framework in a real data integration scenario, involving DBpedia and several biodiversity data providers.
- Chapter 7 summarizes and reviews the presented work, and sketches some perspectives for future research.

Chapter 2

Related Work

The Web of Data is an open environment consisting of heterogeneous, distributed and highly structured information sources. Establishing uniform information access in this kind of setting has become a major research challenge and several data integration approaches [65, 41, 89, 85, 84, 79], or query execution strategies [35, 36, 87, 31, 74] have been proposed. In this thesis we examine certain aspects of data integration in the Semantic Web, including: (*a*) *ontology mapping*, (*b*) *mapping inference*, (*c*) *identification of mapping inconsistencies*, and (*d*) *SPARQL query rewriting* in the context of ontology based mediator architectures. In what follows, we present the most relevant research to the issues addressed in our work.

2.1 Ontology Mapping

Ontology mapping, is the task of relating the vocabulary of two ontologies by specifying a set of correspondences and axioms between ontology terms. Apart from the trivial case, several mapping formalisms involving multiple schemas and adopting different strategies have been proposed in the context of mediator based architectures [50, 51, 21]. The most well-known formalisms in this field are the (*a*) Global-As-View, (*b*) Local-As-View, and (*c*) Global-and-Local-As-View, while the selection of the best depends solely on system requirements.

The *Global-As-View (GAV)* [29, 21] approach is based on the idea that each element of the mediator ontology schema should be characterized in terms of a view over the integrated data source ontologies. In some sense, the mapping explicitly instructs the system how to retrieve the data when evaluating the various elements of the mediator ontology schema. GAV approach is effective whenever the data integration system is based on a set of sources that is stable. Extending the system with a new data source, may have an impact on the mapping definition of the various global ontology elements, whose associated views need to be redefined.

The *Local-As-View (LAV)* [52, 21] formalism takes the opposite approach to GAV. It is based

on the idea that each element of the integrated data source ontology schemas should be characterized in terms of a view over the mediator ontology. LAV approach is effective whenever the data integration system is based on a global ontology schema that is stable and well-established in the organization. Extending the system with a new source simply means the enrichment of mapping with new assertions (without performing any other changes).

The *Global-and-Local-As-View (GLAV)* [54, 21] approach is a combination of the GAV and LAV formalisms. It offers the expressive power of both techniques, aiming to overcome their limitations. GLAV is based on the idea that a view over the integrated data source ontology schemas should be characterized in terms of a view over the mediator ontology.

To the best of our knowledge, there is no system supporting these formalisms in the context of ontology based mediator architectures. SPARQL–RW aiming to satisfy different system requirements and user needs, supports all three aforementioned mapping formalisms by providing a model and a language for the expression of complex ontology mappings.

Mapping discovery is a task that can be achieved either manually, automatically, or semi-automatically. Many recent strategies [17, 3, 27] and tools related to automatic or semi-automatic mapping discovery have been proposed and have their performance analyzed [26, 43, 90, 18, 81, 7]. Although these techniques provide satisfactory results, the quality of auto-generated mappings cannot be compared with manually specified ones. Manual mapping specification is undoubtedly a difficult process, however, it is able to provide declarative and highly expressive correspondences by exploiting domain knowledge expertise.

SPARQL–RW supports a set of rich and flexible mappings types between ontology schemas. Therefore, manual mapping definition and discovery are considered essential for exploiting the mapping model capabilities in their full potential. To assist the mapping discovery task, we have implemented a deductive method for performing *mapping inference* based on [12, 61, 58, 59]. The method exploits model-theoretic semantics and requires an initial set of mappings to be provided in order to perform effectively. It is based on reasoning over the mediator ontology schema, the integrated ontology schemas and the initial mapping set, exploiting any underlying semantics. To this end, the fact that the SPARQL–RW mapping model is based on Description Logics (DL) is considered crucial. To the best of our knowledge only one system [42] implements a similar approach for diagnosing and repairing alignments. Complementary to the adopted method, mappings generated by ontology matching systems can be used effectively after limited post-processing.

Barring a few exceptions [77, 47, 60] the problem of identifying and eliminating mapping inconsistencies has received limited attention in the literature. SPARQL–RW provides built-in functionality for performing *inconsistency identification* in terms of *mapping formalism violations* and *semantic contradictions*. Regarding the latter, it is performed using a similar approach to mapping inference, exploiting the underlying mapping and schema semantics.

For the *mapping representation* task, several different languages have been proposed, including

OWL [57], *C-OWL* [11], *SWRL* [40], MAFRA [55], the *Alignment Format* [25] and its *EDOAL* extension [20]. However, only some of them combine and satisfy the core criteria required for data integration systems; that is, *simplicity*, *expressiveness*, *executability*, and *schema language agnosticism*. Furthermore, to the extent of our knowledge, none of the above languages considers the use of formalisms in mapping representation, while few of them are capable of describing mappings that involve multiple ontology schemas. A thorough comparison of these languages for the task of mapping specification in the context of data integration is available in [26].

2.2 SPARQL Query Rewriting

Query rewriting is a well-known technique, extensively used for addressing various issues including *query optimization*, *query answering*, and performing *information integration*. Although, this method has been studied extensively in databases, it has received limited attention by the Semantic Web community especially for performing query mediation tasks in the context of data integration. To the extent of our knowledge, there is no system performing SPARQL 1.1 query rewriting in general, or SPARQL query rewriting by exploiting any well-known mapping formalism in the context of ontology based mediator architectures.

SPARQL–RW provides a query rewriting method, based on ontology mappings, for performing query mediation over diverse, in terms of schema, federated RDF data sources. Our approach is motivated by view-based query answering techniques [33, 32], applied in databases. While these techniques were exploiting materialized views, SPARQL–RW uses mappings between virtual views; that is, mappings between complex DL based expressions involving schema terms. Virtual views were preferred since they require no additional space and, unlike materialized views, can be modified at any time without any additional cost.

The use of DL semantics in the SPARQL–RW mapping model was inspired by previous approaches [6, 16, 15] dealing with query answering using views in Description Logics. While mappings based on DL require a more complex query rewriting strategy (involving DL to SPARQL translation), their DL nature enables inconsistency identification and mapping inference through the use of reasoning techniques.

Recent studies in the field of query rewriting [45, 46] propose the use of graph based algorithms for achieving scalability in view-based conjunctive query answering. However, these approaches focus on the development of algorithms for improving the rewriting performance when dealing with thousands of views, and do not consider neither ontology mappings or a specific query language. On the contrary, Le et al. [49] presented a native SPARQL query rewriting method for performing view-based query answering. Apart from transferring the problem of view-based query answering from relational databases to RDF, they do not consider certain aspects related to view definition, maintenance and query mediation over diverse federated RDF data sources.

An approach which comes closer to ours, was presented recently by Correndo et al. [19]. Correndo et al. propose a SPARQL query rewriting method for implementing data integration over linked data using mappings between graphs. Although their method looks promising, the use of mappings between graphs seems to restrict the supported query types especially in case of filter expressions containing ontology terms. Furthermore, this choice results to limited mapping expressiveness, weak semantics and high difficulty in mapping definition. In contrast to our proposal, mappings generated by ontology matching systems need heavy post-processing in order to be used in query rewriting.

Lopes et al. [53] presented an approach for rewriting SPARQL queries using heterogeneous mappings. The provided method adopts a rule-based formalism for mapping definition and deals with structural and concept-based heterogeneities, in terms of supporting mappings between a class and a property. However, they do not supply any information about the supported mapping types or their level of complexity, even by comparing the mapping capabilities of their method to others. Furthermore, apart from the fact that they do not provide any proof about the soundness and completeness of their algorithms, their technique seems to consider only data queries and not schema. On the contrary, SPARQL–RW considers both data and schema queries, while it is proved to provide semantics preserving queries with respect to the mapping types supported by the mapping model.

It is worth to note that both previous techniques are limited in exploiting mappings between two ontologies and do not consider any mapping formalism for achieving data integration over multiple data sources. Regarding query rewriting, both methods consider exclusively the case of rewriting a SPARQL query posed over a source ontology ontology, in terms of a target ontology. As a result, in order for these approaches to be used effectively in a data integration scenario that involves multiple data sources, several adaptations need to be made.

Other approaches in the field of query rewriting include those presented by Fujino et al. [28] and Zheng et al. [92]. Fujino et al. study SPARQL query rewriting using incomplete ontology mappings between two ontologies. Nevertheless, the exploited mappings are auto-generated using ontology matching techniques, and therefore, provide limited expressiveness. On the other hand, Zheng et al. proposed a system performing SPARQL query mediation over RDF data sources with disparate contexts. Their query rewriting approach is based exclusively on the exploitation of context mappings, resolving different assumptions on property value interpretations among data sources. To this end, they mainly exploit value transformation functions.

Chapter 3

Background

This chapter presents a brief overview of the standards and technologies used in this thesis. Section 3.1 describes RDF, the main data model for representing information about resources in the World Wide Web. Section 3.2 presents OWL, the standard language for defining and instantiating Web ontologies. Finally, Section 3.3 describes SPARQL, the standard query language for RDF.

3.1 The RDF Data Model

The *Resource Description Framework (RDF)* [56, 13, 37] is the standard data model 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 [23]), and describing resources in terms of simple properties and property values. The atomic constructs of RDF are statements represented as triples of the form *subject-predicate-object*. The *subject* represents the described resource, the *predicate* represents a property and the *object* a property value.

Resources in RDF may also be anonymous and not identified by an IRI. Such resources are called *blank nodes* and can be used both in the subject and object part of an RDF triple. Data values, on the other hand, are represented by the so-called *literals* and can be used only as property values in the object part. The value of every literal is generally described by a sequence of characters, while the interpretation of such sequences is determined based on a datatype URI, usually combined with the actual data value.

Definition 3.1 (RDF triple). Let I be the set of IRIs, L the set of RDF Literals, and B the set of blank nodes. A triple $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$ is called an *RDF triple or RDF statement*, where s, p, and o are a subject, predicate, and object, respectively. □

Graphically, an RDF triple (s, p, o) is represented by a labeled edge $s \xrightarrow{p} o$, while a collection of RDF triples can be intuitively understood as a directed labeled graph, where resources are nodes and statements are arcs connecting the nodes. 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 3.2 (RDF graph). An *RDF graph or RDF dataset* is a finite set of RDF triples. An RDF graph is ground if it has no blank nodes. □

RDF provides a way to express simple statements about resources, using named properties and values. However, RDF user communities also need the ability to define the vocabularies they intend to use in those statements, specifically, to indicate that they are describing specific kinds (or classes) of resources, and use specific properties in describing those resources. RDF itself provides no means for defining such application-specific classes and properties. Instead, such classes and properties are described as an RDF vocabulary, using extensions to RDF provided by the RDF Schema.

RDF Schema (RDFS) [13], is an extension of RDF designed to describe resources, relationships between them and properties (traditional attribute-value pairs). It does not provide a vocabulary of application-specific classes, but the facilities needed to describe such classes and properties, and to indicate how properties and classes are intended to be used together in RDF data. In other words, RDF Schema provides a type system for RDF. The RDF Schema type system is similar in some respects to the type systems of object-oriented programming languages.

An *RDF class* corresponds to the generic concept of a type and can be used to represent almost any category of resources. The elements of a class are known as *instances* of that class. An *RDF property*, on the other hand, can be used to characterize a class (or classes) and describe the relation between subject resources and object resources. Information regarding how properties and classes are intended to be used together is supplied by exploiting constructs that define the domain and range of a property through the use of classes. Moreover, both classes and properties can be organized in hierarchies providing a simple notion of inheritance based on set inclusion.

Finally, the RDF Schema offers a number of built-in properties which can be used to provide documentation and general information about instances. RDF Schema facilities are themselves provided in the form of an RDF vocabulary; that is, as a specialized set of predefined RDF resources with their own special meanings. It is worth to note that RDF semantics is expressed through the mechanism of inferencing; that is, the meaning of any construct in RDF is given by the inferences that can be derived from it. For a detailed description of the RDF semantics refer to [37].

3.2 Web Ontology Language (OWL)

The *Web Ontology Language (OWL)* [57, 64, 63] is the standard language for defining and instantiating Web ontologies. OWL is defined as a vocabulary, such as RDF and RDF Schema, but offers stronger syntax, richer semantics and greater machine interpretability. It was based on Description Logics (DLs), a family of formal knowledge representation languages with attractive and

well-understood computational properties. The initial OWL specification defines three increasingly expressive sublanguages (species) designed for use by specific communities of implementers and users. Each of these languages is an extension of its simpler predecessor, both in what can be legally expressed and in what can be validly concluded.

- *OWL Lite* supports those users primarily needing a classification hierarchy and simple constraints. It has lower formal complexity compared to OWL DL.
- *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 provides all OWL language constructs, under certain restrictions.
- *OWL Full* is meant for users who want maximum expressiveness and the syntactic freedom of RDF with no computational guarantees.

Generally, OWL provides constructs for creating *classes*, *properties*, defining *instances*, as well as applying various operations on them. Minor syntactic differences and construct limitations between the above mentioned sublanguages, are extensively discussed on [57, 38].

An *OWL individual or instance* is an object that corresponds to a *DL individual*. It may belong to none, one or more OWL classes. OWL provides mechanisms in order to declare two individuals to be identical or different, using certain constructors.

An *OWL class* represents a set of individuals that share common properties. It corresponds to a *DL concept* and may contain any number of individuals, instances of the class. A class may be defined to be *subclass* of another, inheriting characteristics from its parent superclass. This corresponds to logical subsumption and DL concept inclusion. Similarly, two classes may be defined to be *equivalent*, indicating that they describe precisely the same instances. This corresponds to logical equivalence and DL concept equality. OWL provides additional constructors which can be used to define complex classes, the so-called *class expressions*. To this end, it supports the basic set operations, namely *union*, *intersection* and *complement*. Additionally, classes can be *enumerated* by specifying explicitly the instance members of a class; that is, the *class extension*. Note that it is possible to assert that class extensions must be disjoint.

A *property* is a directed binary relation that specifies class characteristics and corresponds to a *DL role*. Unlike RDF Schema, OWL distinguishes the properties whose range is a set of individuals from the properties whose range is a set of data values. Thus, *OWL object properties* are relations between class instances, while *OWL datatype properties* are relations between class instances and RDF literals or XML Schema datatypes. A property may be defined to be *subproperty* of another, inheriting characteristics from its parent superproperty. Similarly, two properties may be defined to be *equivalent*, indicating that they describe the same binary relations. Properties may possess logical capabilities such as being *transitive*, *symmetric*, *inverse* and *functional*. Furthermore, apart

from being able to specify the domain and range of a property, it is possible to constrain a property range in specific contexts using either *cardinality* or *value restrictions*. For a detailed description of the OWL semantics refer to [66].

3.2.1 The OWL 2 Standard

The OWL 2 Web Ontology Language (OWL 2) [64, 63] is an extension and revision of the initial Web Ontology Language specification (referred to hereafter as OWL 1). OWL 2 is a W3C recommendation since October 2009. It provides a very similar overall structure with OWL 1, and is backwards compatible with it.

OWL 2 introduces a plethora of new features, some of which are referred as syntactic sugar, since they do not change the expressiveness or the semantics of OWL 1; they have been introduced in order to make some common statements easier to be constructed like *disjoint union of classes*. On the other hand, some other OWL 2 features offer significant expressiveness, including: *keys*; *property chains*; *richer datatypes and data ranges*; *qualified cardinality restrictions*; *asymmetric, reflexive, and disjoint properties*; and *enhanced annotation capabilities*.

OWL 2 defines three new profiles (commonly called fragments or sublanguages in computational logic); that is, trimmed down versions of OWL 2 trading some expressive power for the efficiency of reasoning.

- *OWL 2 EL* is particularly useful in applications employing ontologies that contain very large numbers of properties and/or classes. It allows polynomial time algorithms for all standard inference types, such as satisfiability checking, classification, and instance checking.
- *OWL 2 QL* is aimed at applications that use very large volumes of instance data, and where query answering is the most important reasoning task. It features polynomial time algorithms for all standard inference types.
- *OWL 2 RL* is aimed at applications that require scalable reasoning without sacrificing too much expressive power. The ontology consistency, class expression satisfiability and subsumption, instance checking, and conjunctive query answering problems can be solved in polynomial time with respect to the size of the ontology.

The OWL 2 profiles are defined by placing restrictions on the structure of OWL 2 ontologies and are extensively described on [63]. Finally, it is worth to note that in the OWL 2 specification some of the restrictions applicable to OWL DL have been relaxed; as a result, the set of RDF Graphs that can be handled by DL reasoners is slightly larger in OWL 2.

3.3 SPARQL Query Language

The SPARQL Query Language [73] is the standard language for querying RDF data. The main body of the query, is a complex RDF graph pattern expression that may include RDF triples with variables, conjunctions, disjunctions, optional parts, and constraints over the values of the variables. The head of the query, is an expression that indicates how to construct the answer to the query. The evaluation of a SPARQL query over an RDF dataset is based on *graph pattern matching*.

In what follows, we provide an overview of the core syntax (Section 3.3.1) and semantics (Section 3.3.2) of SPARQL. Moreover, Section 3.3.3 presents the SPARQL 1.1 Standard, while Table 3.1 summarizes the notation that we adopt, along with a brief description.

Table 3.1: The notation used in the definition of SPARQL syntax and semantics.

Notation	Description
I	The set of IRIs.
L	The set of RDF Literals.
B	The set of blank nodes.
V	The set of variables.
ω	A graph pattern solution (or simply solution) $\omega : V \rightarrow (I \cup B \cup L)$.
$dom(\omega)$	The domain of a graph pattern solution ω (subset of V).
$var(x)$	The variables occurring in a graph pattern or built-in condition x.
$\omega(P)$	The graph obtained by replacing the variables in graph pattern P according to a graph pattern solution ω (abusing notation).
$\omega \models R$	A graph pattern solution ω satisfies a built-in condition R.
$[[\cdot]]$	Evaluation function.
$\bowtie$	Join operator.
$⟕$	Left outer join operator.
$\setminus$	Difference operator.
$\pi_{\{\dots\}}$	Projection operator.
$\cup$	Union operator.
$\cap$	Intersection operator.
$bound$	SPARQL unary predicate.
AND	Binary operator that corresponds to the SPARQL conjunction construct.
OPT	Binary operator that corresponds to the SPARQL `OPTIONAL` construct.
UNION	Binary operator that corresponds to the SPARQL `UNION` construct.
FILTER	Binary operator that corresponds to the SPARQL `FILTER` construct.
$\neg, \vee, \wedge$	Logical not, or, and.
$<, \leq, \geq, >, =$	Inequality/equality operators.

3.3.1 Syntax of SPARQL

In this section we present an overview of the SPARQL syntax. Let V be an infinite set of variables disjoint from the sets of IRIs (I), RDF Literals (L) and blank nodes (B). We assume that the elements from V are prefixed by the symbol '?'.

Definition 3.3 (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. For a given triple pattern t we use $var(t)$ to denote the set of variables occurring in the components of t. □

Let AND, OPT, UNION and FILTER be binary operators that correspond to SPARQL conjunction, `OPTIONAL`, `UNION`, and `FILTER` constructs, respectively.

Definition 3.4 (Built-in condition). A SPARQL *built-in condition* is a boolean expression composed by logical connectives ($\neg$, $\vee$, $\wedge$), inequality/equality symbols ($<$, $\leq$, $\geq$, $>$, $=$), and unary predicates like $bound$, $isBlank$, and $isIRI$, over constants and elements of the set $I \cup L \cup V$. For a given built-in condition R we use $var(R)$ to denote the set of variables occurring in R. □

Definition 3.5 (Filter expression). The built-in condition used alongside a `FILTER` construct in order to restrict the query solutions is called *filter expression*. □

Definition 3.6 (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 the expressions P_1 AND P_2, P_1 OPT P_2, and P_1 UNION P_2 are graph patterns (conjunction graph pattern, optional graph pattern, and union graph pattern, respectively).
- If P is a graph pattern and R is a built-in condition, then the expression P FILTERR is a graph pattern (a filter graph pattern).

For a given graph pattern P, $var(P)$ denotes the set of variables occurring in P. □

Definition 3.7 (Basic graph pattern). A finite sequence of conjunctive triple patterns and possible filters is called *basic graph pattern*. □

SPARQL allows four query types: `SELECT`, `ASK`, `CONSTRUCT` and `DESCRIBE`. These query types use the solutions provided from pattern matching to form result sets or RDF graphs.

Definition 3.8 (Basic `SELECT` query form). Let P be a graph pattern and $(v_1, \ldots, v_n) \subseteq var(P)$ an ordered list of variables. The basic syntax of a SPARQL `SELECT` query is defined as follows: `SELECT` $(v_1, \ldots, v_n)$ `WHERE` (P). □

The `SELECT` query form returns a solution sequence; that is, a sequence of variables along with their bindings. The `ASK` query form returns no information about the possible query solutions, just whether or not a solution exists. The `CONSTRUCT` query form returns an RDF graph structured according to a provided graph pattern template, while the `DESCRIBE` query form returns an RDF graph providing a "description" of the matching resources.

Example 3.1 ***(Select Query, Conjunct./Optional Graph Pattern, Filter).*** Assume the following RDF dataset providing information about books, authors and publishers.

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

_:a  ns:title      "Database Systems" .
_:a  ns:author     _:e .
_:a  ns:publisher  "Prentice Hall".
_:b  ns:title      "CS Foundations" .
_:b  ns:author     _:e .
_:c  ns:title      "Compilers: Principles and Techniques" .
_:d  ns:title      "Introduction to Algorithms" .
_:d  ns:author     _:f .
_:e  ns:name       "Jeffrey D. Ullman" .
_:f  ns:name       "Thomas H. Cormen" .
```

Moreover, consider the following query: "Return book titles written by `Jeffrey D. Ullman` and optionally their publisher.". The SPARQL syntax of this query is depicted below.

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

SELECT ?title ?publisher
WHERE {
   ?book   ns:title  ?title .
   ?book   ns:author ?author .
   ?author ns:name   ?name .
   OPTIONAL {?book ns:publisher ?publisher}
   FILTER (?name = "Jeffrey D. Ullman")
}
```

The query combines a sequence of conjunctive triple patterns followed by an optional graph pattern and a filter restriction on author names. After evaluating the query over the aforementioned RDF dataset, we retrieve the results presented in the following table.

?title	?publisher
"Database Systems"	"Prentice Hall"
"CS Foundations"	

Consider that in an optional match, either the optional graph pattern matches a graph, thereby defining and adding bindings to one or more solutions (this is the case of the first query solution), or it leaves a solution unchanged without adding any additional bindings (this is the case of the second query solution). The book `"Compilers: Principles and Techniques"` has been discarded from the results since there is no information regarding its author, and thus no match for the second triple pattern of the query. On the other hand, although the book `"Introduction`

to Algorithms" provides information about its author, thus matching the three conjunctive triple patterns, it has been discarded since its author's name value fails to satisfy the filter restriction. □

SPARQL provides various *solution sequence modifiers* which can be applied on the initial solution sequence. The supported solution sequence modifiers are: DISTINCT, REDUCED, LIMIT, OFFSET, and ORDER BY. The DISTINCT modifier ensures that duplicate solutions are eliminated from the solution set. On the other hand, REDUCED simply allows duplicate solutions to be reduced. The LIMIT modifier is used to put an upper bound on the number of solutions returned, while OFFSET causes the generated solutions to start after a specified number of solutions. Finally, ORDER BY is used to establish the order of a solution sequence.

Example 3.2 ***(Select Query, Union Graph Pattern, Sequence Modifiers).*** Assume the following RDF dataset that provides information about books, categories and citations.

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

_:a  ns:title      "Database Systems" .
_:a  ns:category   ns:DM .
_:a  ns:citations  1257 .
_:b  ns:title      "Introduction to IR" .
_:b  ns:category   ns:DM .
_:b  ns:citations  1175 .
_:c  ns:title      "AI: A Modern Approach" .
_:c  ns:category   ns:AI .
_:c  ns:citations  950 .
_:d  ns:title      "The Description Logic Handbook" .
_:d  ns:category   ns:AI .
_:d  ns:citations  1342 .
_:e  ns:title      "Modern Operating Systems" .
_:e  ns:category   ns:OS .
_:e  ns:citations  3195 .
```

Moreover, consider the following query: "Return the 3 most cited books in Data Management (DM) and Artificial Intelligence (AI) along with the number of their bibliographic citations. The results should be distinct and formed in descending order based on citation values.". The SPARQL syntax of this query is depicted below.

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

SELECT DISTINCT ?title ?citations
WHERE {
   ?book ns:title     ?title .
   ?book ns:citations ?citations .
   {?book ns:category ns:DM}
```

```
   UNION
   {?book ns:category ns:AI}
} ORDER BY DESC (?citations) LIMIT 3
```

The query combines a sequence of conjunctive triple patterns followed by a union graph pattern that matches book instances in the Data Management and Artificial Intelligence categories. It also contains several solution sequence modifiers including `DISTINCT`, `ORDER BY` and `LIMIT`. After evaluating the query over the aforementioned RDF dataset, we retrieve the results presented in the following table.

?title	?citations
"Database Systems"	1342
"The Description Logic Handbook"	1257
"Introduction to IR"	1175

Consider that solution sequence modifiers are applied after the evaluation of the query's graph pattern over the RDF dataset. As a result, although the RDF graph of book `"AI: A Modern Approach"` matches the query graph pattern and qualifies to the initial solution sequence, it is discarded after sorting the solutions and restricting their number to 3. On the other hand, the RDF graph of book `"Modern Operating Systems"` failed to match the query graph pattern, and thus qualify to the initial solution sequence, since it does not contain any property `ns:category` with values `ns:DM` or `ns:AI`. Finally, note that the `DISTINCT` modifier ensures that solutions in the sequence are unique. □

For a detailed description of the SPARQL syntax, as well as a complete set of illustrative examples, refer to [73].

3.3.2 Semantics of SPARQL

In this section we provide an overview of the semantics of SPARQL based on [67], considering a function-based representation of a SPARQL graph pattern solution.

Definition 3.9 (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 ω, $dom(\omega)$, is the subset of V where ω is defined. The empty graph pattern solution $\omega_\emptyset$ is the graph pattern solution with empty domain. The result of the evaluation of a SPARQL graph pattern over an RDF dataset is a set Ω of graph pattern solutions ω. □

Two graph pattern solutions ω_1 and ω_2 are compatible when for all $x \in dom(\omega_1) \cap dom(\omega_2)$, 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 $\omega_\emptyset$ is compatible with any other graph pattern solution.

Let Ω_1 and Ω_2 be sets of graph pattern solutions and $\mathcal{J}$ be a set of variables. The *join*, *union*, *difference*, *projection* and *left outer join* operations between Ω_1 and Ω_2 are defined as follows:

$$\begin{aligned}
\Omega_1 \bowtie \Omega_2 &= \{\omega_1 \cup \omega_2 \mid \omega_1 \in \Omega_1,\ \omega_2 \in \Omega_2 \text{ are compatible graph pattern solutions}\} \\
\Omega_1 \cup \Omega_2 &= \{\omega \mid \omega \in \Omega_1 \text{ or } \omega \in \Omega_2\} \\
\Omega_1 \setminus \Omega_2 &= \{\omega \in \Omega_1 \mid \text{ for all } \omega' \in \Omega_2,\ \omega \text{ and } \omega' \text{ are not compatible}\} \\
\pi_{\mathcal{J}}(\Omega_1) &= \{\omega \mid \omega' \in \Omega_1,\ dom(\omega) = dom(\omega') \cap \mathcal{J},\ \forall x \in dom(\omega) : \omega(x) = \omega'(x)\} \\
\Omega_1 \mathbin{⟕} \Omega_2 &= (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2)
\end{aligned}$$

The semantics of SPARQL is defined as a function $[[\cdot]]_D$ which takes a graph pattern expression or a SPARQL query and an RDF dataset D and returns a set of graph pattern solutions. Definition 3.10 specifies the satisfiability of a filter expression, while Definitions 3.11 and 3.12 specify the evaluation of a graph pattern expression and a basic `SELECT` SPARQL query, respectively, over an RDF dataset.

Definition 3.10 (Filter expression satisfiability). Let v_1, v_2 be variables, c a constant or an element of the set $I \cup L \cup V$, and $\diamond$ a symbol of the set $\{<, \leq, \geq, >, =\}$. Given a graph pattern solution ω and a built-in condition R, we say that ω satisfies R, denoted by $\omega \models R$, if:

1. R is of type $bound(v_1)$ and $v_1 \in dom(\omega)$;
2. R is of type $(v_1 \diamond c)$, $v_1 \in dom(\omega)$ and $\omega(v_1) \diamond c$;
3. R is of type $(v_1 \diamond v_2)$, $v_1 \in dom(\omega)$, $v_2 \in dom(\omega)$ and $\omega(v_1) \diamond \omega(v_2)$;
4. R is of type $(\neg R_1)$, R_1 is a built-in condition, and it is not the case that $\omega \models R_1$;
5. R is of type $(R_1 \vee R_2)$, R_1, R_2 are built-in conditions, and $\omega \models R_1$ or $\omega \models R_2$;
6. R is of type $(R_1 \wedge R_2)$, R_1, R_2 are built-in conditions, $\omega \models R_1$ and $\omega \models R_2$. □

Definition 3.11 (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 filter expression. The evaluation of a graph pattern over D, denoted by $[[\cdot]]_D$, is defined recursively as follows:

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

Definition 3.12 (Basic `SELECT` query evaluation). Let P be a SPARQL graph pattern and let $(v_1, \ldots, v_n) \subseteq var(P)$ be an ordered list of variables. The evaluation of a basic SPARQL `SELECT` query Q over D, denoted by $[[\cdot]]_D$, is defined as follows:

6. $[[Q]]_D = [[\texttt{SELECT}\ (v_1, \ldots, v_n)\ \texttt{WHERE}\ (P)]]_D = \pi_{v_1,\ldots,v_n}([[P]]_D)$ □

For a detailed description of SPARQL semantics and for a complete set of illustrative examples related to the evaluation of a graph pattern, refer to [67, 68].

3.3.3 The SPARQL 1.1 Standard

The SPARQL 1.1 Standard is the result of the W3C SPARQL Working Group on the extension of the SPARQL Query Language in several aspects. To this end, SPARQL 1.1 can be considered as a set of specifications that provide languages and protocols to query and manipulate RDF graph content on the Web or in an RDF store. The standard comprises, among others, the Query Language, the Federated Query, and the Update Language specifications which are briefly described below.

The *SPARQL 1.1 Query Language* [34, 4] adds a number of new features to the previous SPARQL specification (referred to hereafter as SPARQL 1.0) including *path expressions*, *aggregate functions*, *value assignment* and *nested queries*.

Path expressions or property paths are possible routes through a graph between two graph nodes. They allow for more concise expressions for some SPARQL basic graph patterns and they also add the ability to match connectivity of two resources by an arbitrary length path. Property paths are formed using IRIs or prefixed names along with binary operators (`/`, `|`), unary operators (`^`, `*`, `+`, `?`, `!`) and brackets for specifying complex group paths. Table 3.2 presents the semantics of the aforementioned operators and the supported property path syntax forms.

Table 3.2: SPARQL 1.1 property path syntax. Consider `elt` as a path element, which may itself be composed of path constructs.

Syntax Form	Matches
`iri`	An IRI. A path of length one.
`^elt`	Inverse path (object to subject).
$\texttt{elt}_1\texttt{/elt}_2$	A sequence path of `elt1` followed by `elt2`.
$\texttt{elt}_1\texttt{|elt}_2$	A alternative path of `elt1` or `elt2` (all possibilities are tried).
`elt*`	A path that connects the subject and object of the path by zero or more matches of `elt`.
`elt+`	A path that connects the subject and object of the path by one or more matches of `elt`.
`elt?`	A path that connects the subject and object of the path by zero or one matches of `elt`.
`!iri` or $\texttt{!(iri}_1\texttt{|...|iri}_n\texttt{)}$	Negated property or negated property set. An IRI which is not one of $\texttt{iri}_i$. `!iri` is short for `!(iri)`. Negated properties where the excluded matches are based on reversed paths may also appear (`!^iri`).
`(elt)`	A group path `elt`, brackets control precedence.

Aggregates apply expressions over groups of solutions. By default a solution set consists of a single group, containing all solutions. Groupings may be specified using the `GROUP BY` syntax, while the supported aggregates include: `COUNT`, `SUM`, `MIN`, `MAX`, `AVG`, `GROUP_CONCAT`, and `SAMPLE`.

Value assignment allows the value of an expression to be added to a solution mapping by binding a new variable to the value of the expression. The variable can then be used in the query and also can be returned in results. Value assignment has the form (`expression AS ?var`) and can be applied in the `SELECT` clause, in the `GROUP BY` clause, or in the query graph pattern using the `BIND` keyword. If the evaluation of the expression produces an error, the variable remains unbound for that solution but the query evaluation continues. Data can also be directly included in a query using `VALUES` for inline data.

Example 3.3 ***(Select Query, Property Paths, Value Assignment).*** Consider the following query: "Return book titles written by `Franz Baader` along with their prices after taking into account the discount.". The SPARQL syntax of this query is depicted below.

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

SELECT ?title (?p*(1-?discount) AS ?price)
WHERE {
   ?book ns:title    ?title .
   ?book ns:price    ?p .
   ?book ns:discount ?discount .
   ?book ns:author/ns:name ?name .
   FILTER (?name = "Franz Baader")
}
```

This example demonstrates both the use of property paths and value assignment. A property path expression is used to connect directly book instances and author names in order for the later to be restricted to `"Franz Baader"` values. This could have been expressed using a conjunction of two triple patterns, connecting books with authors and then authors with names, as shown in the Example 3.1. Finally, value assignment is used in the `SELECT` clause, where final book prices are calculated, and subsequently binded to a new variable. □

Nested queries or sub-queries can be considered as a way to embed queries within other queries in order to tackle certain cases, such as limiting the number of results from some sub-expression within the query. Due to the bottom-up nature of SPARQL query evaluation, sub-queries are evaluated logically first, and the results are projected up to the outer query.

Moreover, it is worth to note that SPARQL 1.1 incorporates two styles of *negation* by introducing the `NOT EXISTS` and `MINUS` operators. `NOT EXISTS` is based on the idea of testing whether a pattern exists in the data, given the bindings already determined by the query pattern. On the other hand, `MINUS` relies on removing matches based on the evaluation of two patterns.

Example 3.4 ***(Select Query, Nested Queries, Aggregates, Value Assignment, Sequence Modifiers).*** Consider the following query: "Return the 10 most cited authors along with their number of

bibliographic citations. The results should be formed in descending order based on total citation values.". The SPARQL syntax of this query is depicted below.

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

SELECT ?name ?total
WHERE {
   ?author ns:name ?name .
   {
      SELECT ?author (SUM(?citations) AS ?total)
      WHERE {
         ?book ns:author    ?author .
         ?book ns:citations ?citations .
      } GROUP BY ?author
   }
} ORDER BY DESC (?total) LIMIT 10
```

The inner query calculates total citations per author, after taking into consideration authors' individual book citations. Subsequently, the outer query enriches the retrieved solution sequence with author names and sorts the solutions in descending order based on the total citation value. The number of solutions in the final solution sequence is restricted to 10. In aggregate queries and sub-queries, variables that appear in the query pattern, but are not in the `GROUP BY` clause, can only be projected or used in select expressions if they are aggregated. That is, any variables that are not aggregated must appear inside a `GROUP BY` clause. Finally, note that only variables projected out of the sub-query are visible, or in scope, to the outer query. □

The *SPARQL 1.1 Federated Query* [72, 14] is an extension of the SPARQL 1.1 Query Language for executing queries distributed over different SPARQL endpoints. The growing number of SPARQL query services offer data consumers an opportunity to merge data distributed across the Web. Using the `SERVICE` keyword extension, a user or an application may instruct the federated query processor to invoke a portion of a SPARQL query against a remote SPARQL endpoint. Results are returned to the federated query processor and are combined with results from the rest of the query.

Finally, the *SPARQL 1.1 Update* [30] is a companion language and envisaged to be used in conjunction with the SPARQL 1.1 Query Language. It provides several facilities for specifying and executing updates to RDF graphs in a graph store, including operators to: (*a*) *insert* new triples into an RDF graph, (*b*) *delete* triples from an RDF graph, (*c*) *load* an RDF graph into the graph store, (*d*) *clear* an RDF graph in the graph store, (*e*) *create* a new RDF graph, (*f*) *drop* an RDF graph, (*g*) *copy*, *move*, or *add* the content of one RDF graph to another, and (*h*) perform a *group of update operations* as a single action.

Example 3.5 ***(Select Query, Federated Query).*** Consider the following query: "Return the authors

of the book `Database Systems`, as well as information about their nationality and optionally their institution.". The SPARQL syntax of this query is depicted below.

```
@PREFIX ns: <http://example.org/books#>
@PREFIX dbpedia: <http://dbpedia.org/ontology/>

SELECT ?name ?nationality ?institution
FROM <http://example.org/books.rdf>
WHERE {
   ?book   ns:title  ?title .
   ?book   ns:author ?author .
   ?author ns:name   ?name .
   FILTER (?title = "Database Systems")
   SERVICE <http://dbpedia.org/sparql> {
      ?author dbpedia:nationality ?nationality .
      OPTIONAL {?author dbpedia:institution ?institution}
   }
}
```

Note that the query accesses a remote SPARQL endpoint, using the `SERVICE` keyword, in order to obtain information about author's nationality and institution, since this kind of data are not available in the local RDF dataset. The results returned from the remote SPARQL endpoint are joined with the data from the local RDF dataset in order to form the final query answer. □

Chapter 4

Mapping Model

In data integration systems, schema mappings are extensively used for performing various tasks under different contexts. In mediator based architectures, for example, schema mappings describe the relationship between the mediated schema and the schemas of the integrated sources. When a query is formulated in terms of the mediated schema, the mediator uses the mappings to reformulate the query into appropriate queries on the sources. Similarly, in data exchange and data warehousing, schema mappings express the relationship between the source and the target database or warehouse. In this context, schema mappings are used to interpret the transmitted data, and in some cases to transform them in order to conform to a single schema; that is, the warehouse schema.

In this chapter, we present the model adopted by the SPARQL–RW Framework for the expression of mappings between ontology schemas in the field of ontology based mediator architectures. In this context, any mapping model, apart from supporting the mapping definition and query rewriting processes, needs to satisfy a number of strong requirements including *mapping expressiveness*, *mapping maintenance*, *query rewriting efficiency*, and *extensibility to new sources*.

Mapping requirements vary from one data integration system to another. Systems integrating highly overlapping sources, in terms of their schema and their underlying data, generally require greater mapping expressiveness compared to systems with minor overlap. Similarly, systems focusing in providing exact and complete answers in client queries, need higher mapping flexibility compared to systems focusing in complete results or approximations.

On the other hand, mapping maintenance is an extremely painful process, especially for systems integrating volatile sources in terms of their schema. To this end, several approaches have been proposed, including automatic identification of mapping inconsistencies and mapping inference through reasoning techniques. The trade-off in using them is that highly expressive mappings cannot be supported, since they may lead to undecidability (computations are not guaranteed to finish in finite time).

Regarding the query rewriting efficiency, it is directly dependent on the soundness and com-

pleteness of the provided algorithms. Since query rewriting in data integration systems is performed through mapping exploitation, the properties characterizing a rewriting algorithm are based on the adopted mapping formalisms and the supported mapping relationships (equivalence/subsumption). Moreover, this requirement is often at odds with that of expressiveness, because more expressive mappings are typically harder to reason about.

Additionally to the aforementioned issues, extensibility to new sources is a basic requirement for data integration systems that are not always based on a stable set of sources. Therefore, supporting mapping formalisms that make the addition/removal of a data source easy, without requiring the inspection of all the other sources, is fundamental for certain cases.

The SPARQL–RW mapping model addresses the above requirements by supporting a set of rich and flexible mapping types between OWL ontologies, as well as notable mapping formalisms, including *Global-As-View (GAV)*, *Local-As-View (LAV)*, and *Global-and-Local-As-View (GLAV)*. Furthermore, it defines a mapping language capable of representing all the specified types of inter-schema correspondences. The model is based on Description Logic semantics, enabling mapping inference and inconsistency identification in a given set of mappings and ontology schemas.

Description Logics (DL) [5] is a well-known family of knowledge representation languages that can be used to represent the knowledge of an application domain in a structured and formally well-understood way. It has been selected as the basis of the SPARQL–RW mapping model since it provides *high expressiveness*, *well-defined semantics*, and *inference capabilities*. These features are highly important for any mapping model aiming to support an ontology based mediator system, in terms of *mapping definition*, *mapping maintenance*, and *query rewriting*.

Mapping definition is supported by the fact that OWL ontology languages build upon the basic elements, constructs and axioms of DL. To this end, serving as the basis of highly expressive languages like OWL 2, DL is capable of describing any relation between complex ontology constructs. Moreover, it provides the means for performing mapping inference and automatic identification of mapping inconsistencies, through the application of reasoning techniques. Both these features are considered invaluable for supporting mapping maintenance and providing assistance in mapping definition. Regarding query rewriting, DL expressions can be directly transformed to SPARQL graph patterns. Therefore, ontology mappings based on DL constructs can be easily exploited by any SPARQL query rewriting algorithm.

Section 4.1 provides the abstract syntax and semantics of the mapping types supported by the SPARQL–RW mapping model. Similarly, Section 4.2 presents the supported mapping formalisms. Section 4.3 demonstrates the expressiveness of SPARQL–RW mapping model by providing a feature comparison with common knowledge representation languages. Sections 4.4 and 4.5 propose methods for performing mapping inference and inconsistency identification in the context of SPARQL–RW mapping model. Finally, Section 4.6 defines the language used for mapping specification and provides several mapping representation examples.

4.1 Abstract Syntax and Semantics

In this section, we present the abstract syntax and semantics of the mapping types supported by the SPARQL–RW mapping model. To this end, we use well-known Description Logic constructors and axioms that deal with *concepts*, *roles* and *individuals*.

The mapping types supported by the SPARQL–RW mapping model build upon elements of four basic types extensively used in ontology schema definition: (*a*) *classes*, (*b*) *object properties*, (*c*) *datatype properties*, and (*d*) *individuals*. These element types comprise the basic notions of OWL, the standard language for defining and instantiating Web ontologies. We treat ontology classes as DL concepts, ontology properties as DL roles and class instances as DL individuals.

Moreover, we utilize concrete domains for illustrating mappings that involve restrictions on property values. In Description Logics, *concrete domains* are used for defining value domains and functions operating upon them. A concrete domain $\mathcal{D}$ consists of a set of values $\Delta^{\mathcal{D}}$, the domain of $\mathcal{D}$, and a set $pred(\mathcal{D})$, the predicate names of $\mathcal{D}$. Each predicate name $P \in pred(\mathcal{D})$ is associated with an arity n, and an n-ary predicate $P^{\mathcal{D}} \subseteq (\Delta^{\mathcal{D}})^n$.

In order to assign meaning and define the semantics of symbols used in the context of the SPARQL–RW mapping model, we assume an interpretation $\mathcal{I}$ consisting of two non-empty sets: (*a*) $\Delta^{\mathcal{I}}$*, the domain of individuals*, and (*b*) $\Delta^{\mathcal{I}}_{\mathrm{D}}$*, the domain of data values*. Moreover, we consider an interpretation function which assigns: (*a*) to every class C a set $C^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$, (*b*) to every data range D a set $D^{\mathrm{D}} \subseteq \Delta^{\mathcal{I}}_{\mathrm{D}}$, (*c*) to every object property R a binary relation $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$, (*d*) to every datatype property U a binary relation $U^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}_{\mathrm{D}}$, (*e*) to every individual o an element $o^{\mathcal{I}} \in \Delta^{\mathcal{I}}$, and (*f*) to every data value v an element $v^{\mathcal{I}} = v^{\mathrm{D}}$.

Tables 4.1, 4.2 and 4.3 provide the syntax and semantics of class and property constructors used in mapping type definition. Similarly, Table 4.4 presents the supported concrete domains and built-in data ranges, along with the binary and unary predicates operating on their values. Apart from built-in data ranges, the SPARQL–RW mapping model allows the specification of custom data ranges and the use of value transformations in mapping definition. Table 4.5 provides an overview of the supported value transformations.

Definition 4.1 (Data Range). A *data range* D is a set of data values. In the context of the SPARQL–RW mapping model it is defined by specifying either a built-in data range along with an optional complex value condition $cond$ (using *and* - $\wedge$, *or* - $\vee$, *not* - ! operators), or a complex value condition alone. The built-in data ranges supported by the model are presented in Table 4.4. Let dr be a built-in data range and P a unary predicate. A custom data range is recursively defined as follows:

$$D \rightarrow dr \mid dr(cond) \mid (cond) \tag{4.1}$$

$$cond \rightarrow P \mid cond \wedge cond \mid cond \vee cond \mid !\, cond \tag{4.2}$$

□

Table 4.1: Class constructors used in mapping definition.

Name	Syntax	Semantics
Atomic	C	$C^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
Intersection	$C_1 \sqcap C_2$	$C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}}$
Union	$C_1 \sqcup C_2$	$C_1^{\mathcal{I}} \cup C_2^{\mathcal{I}}$
Difference	$C_1 - C_2$	$C_1^{\mathcal{I}} \setminus C_2^{\mathcal{I}}$
Existential quantific.	$\exists R.C$	$\{\alpha \in \Delta^{\mathcal{I}} \mid \exists b.\, (\alpha, b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\}$
	$\exists U.D$	$\{\alpha \in \Delta^{\mathcal{I}} \mid \exists b.\, (\alpha, b) \in U^{\mathcal{I}} \wedge b \in D^{\mathcal{D}}\}$
Existential predicate restriction	$\exists (R_1, R_2).P$	$\{\alpha \in \Delta^{\mathcal{I}} \mid \exists b_1, b_2.\, (\alpha, b_1) \in R_1^{\mathcal{I}} \wedge (\alpha, b_2) \in R_2^{\mathcal{I}} \wedge \wedge (b_1, b_2) \in P^{\mathcal{D}}\}$
	$\exists (U_1, U_2).P$	$\{\alpha \in \Delta^{\mathcal{I}} \mid \exists b_1, b_2.\, (\alpha, b_1) \in U_1^{\mathcal{I}} \wedge (\alpha, b_2) \in U_2^{\mathcal{I}} \wedge \wedge (b_1, b_2) \in P^{\mathcal{D}}\}$
Unqualified number restriction	$\geq n\, R$	$\{\alpha \in \Delta^{\mathcal{I}} \mid \lvert\{b \in \Delta^{\mathcal{I}} \mid (\alpha, b) \in R^{\mathcal{I}}\}\rvert \geq n\}$
	$\leq n\, R$	$\{\alpha \in \Delta^{\mathcal{I}} \mid \lvert\{b \in \Delta^{\mathcal{I}} \mid (\alpha, b) \in R^{\mathcal{I}}\}\rvert \leq n\}$
	$= n\, R$	$\{\alpha \in \Delta^{\mathcal{I}} \mid \lvert\{b \in \Delta^{\mathcal{I}} \mid (\alpha, b) \in R^{\mathcal{I}}\}\rvert = n\}$
	$\geq n\, U$	$\{\alpha \in \Delta^{\mathcal{I}} \mid \lvert\{b \in \Delta^{\mathcal{D}} \mid (\alpha, b) \in U^{\mathcal{I}}\}\rvert \geq n\}$
	$\leq n\, U$	$\{\alpha \in \Delta^{\mathcal{I}} \mid \lvert\{b \in \Delta^{\mathcal{D}} \mid (\alpha, b) \in U^{\mathcal{I}}\}\rvert \leq n\}$
	$= n\, U$	$\{\alpha \in \Delta^{\mathcal{I}} \mid \lvert\{b \in \Delta^{\mathcal{D}} \mid (\alpha, b) \in U^{\mathcal{I}}\}\rvert = n\}$
Qualified number restriction	$\geq n\, R.C$	$\{\alpha \in \Delta^{\mathcal{I}} \mid \lvert\{b \in \Delta^{\mathcal{I}} \mid (\alpha, b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\}\rvert \geq n\}$
	$\leq n\, R.C$	$\{\alpha \in \Delta^{\mathcal{I}} \mid \lvert\{b \in \Delta^{\mathcal{I}} \mid (\alpha, b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\}\rvert \leq n\}$
	$= n\, R.C$	$\{\alpha \in \Delta^{\mathcal{I}} \mid \lvert\{b \in \Delta^{\mathcal{I}} \mid (\alpha, b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\}\rvert = n\}$
	$\geq n\, U.D$	$\{\alpha \in \Delta^{\mathcal{I}} \mid \lvert\{b \in \Delta^{\mathcal{D}} \mid (\alpha, b) \in U^{\mathcal{I}} \wedge b \in D^{\mathcal{D}}\}\rvert \geq n\}$
	$\leq n\, U.D$	$\{\alpha \in \Delta^{\mathcal{I}} \mid \lvert\{b \in \Delta^{\mathcal{D}} \mid (\alpha, b) \in U^{\mathcal{I}} \wedge b \in D^{\mathcal{D}}\}\rvert \leq n\}$
	$= n\, U.D$	$\{\alpha \in \Delta^{\mathcal{I}} \mid \lvert\{b \in \Delta^{\mathcal{D}} \mid (\alpha, b) \in U^{\mathcal{I}} \wedge b \in D^{\mathcal{D}}\}\rvert = n\}$
Nominal	o	$o^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}\ with\ \mid o^{\mathcal{I}} \mid = 1$

Table 4.2: Object property constructors used in mapping definition.

Name	Syntax	Semantics
Atomic	R	$R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
Intersection	$R_1 \sqcap R_2$	$R_1^{\mathcal{I}} \cap R_2^{\mathcal{I}}$
Union	$R_1 \sqcup R_2$	$R_1^{\mathcal{I}} \cup R_2^{\mathcal{I}}$
Difference	$R_1 - R_2$	$R_1^{\mathcal{I}} \setminus R_2^{\mathcal{I}}$
Composition	$R_1 \circ R_2$	$\{(\alpha, c) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid \exists b.\, (\alpha, b) \in R_1^{\mathcal{I}} \wedge (b, c) \in R_2^{\mathcal{I}}\}$
Inversion	R^{-}	$\{(b, \alpha) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid (\alpha, b) \in R^{\mathcal{I}}\}$
Transitive closure	R^{+}	$\bigcup_{n \geq 1} (R^{\mathcal{I}})^n$
	R^{*}	$\bigcup_{n \geq 0} (R^{\mathcal{I}})^n$
Existential predicate restriction	$\exists (R_1)(R_2).P$	$\{(\alpha, b) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid \exists c_1, c_2.\, (\alpha, c_1) \in R_1^{\mathcal{I}} \wedge \wedge (b, c_2) \in R_2^{\mathcal{I}} \wedge (c_1, c_2) \in P^{\mathcal{D}}\}$
	$\exists (U_1)(U_2).P$	$\{(\alpha, b) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid \exists c_1, c_2.\, (\alpha, c_1) \in U_1^{\mathcal{I}} \wedge \wedge (b, c_2) \in U_2^{\mathcal{I}} \wedge (c_1, c_2) \in P^{\mathcal{D}}\}$
Domain restr.	$R \upharpoonright C$	$\{(\alpha, b) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid (\alpha, b) \in R^{\mathcal{I}} \wedge \alpha \in C^{\mathcal{I}}\}$
Range restr.	$R \downharpoonright C$	$\{(\alpha, b) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid (\alpha, b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\}$

Table 4.3: Datatype property constructors used in mapping definition.

Name	Syntax	Semantics
Atomic	U	$U^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}_{\mathcal{D}}$
Intersection	$U_1 \sqcap U_2$	$U_1^{\mathcal{I}} \cap U_2^{\mathcal{I}}$
Union	$U_1 \sqcup U_2$	$U_1^{\mathcal{I}} \cup U_2^{\mathcal{I}}$
Difference	$U_1 - U_2$	$U_1^{\mathcal{I}} \setminus U_2^{\mathcal{I}}$
Composition	$R \circ U$	$\{(\alpha, c) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}_{\mathcal{D}} \mid \exists b.\, (\alpha, b) \in R^{\mathcal{I}} \wedge (b, c) \in U^{\mathcal{I}}\}$
Domain restriction	$U \upharpoonright C$	$\{(\alpha, b) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}_{\mathcal{D}} \mid (\alpha, b) \in U^{\mathcal{I}} \wedge \alpha \in C^{\mathcal{I}}\}$
Range restriction	$U \downharpoonright D$	$\{(\alpha, b) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}_{\mathcal{D}} \mid (\alpha, b) \in U^{\mathcal{I}} \wedge b \in D^{\mathcal{D}}\}$
Transformation	$trans(U)$	$\{(\alpha, c) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}_{\mathcal{D}} \mid \exists b.\, (\alpha, b) \in U^{\mathcal{I}} \wedge\ c = trans(b)\}$

Table 4.4: The supported concrete domains ($\mathcal{D}$) and built-in data ranges ($\Delta^{\mathcal{D}}$), along with the binary and unary predicates operating on their values. Consider a value $n \in \Delta^{\mathcal{D}}$.

Concrete Domain	Data Range	Binary Predicates	Unary Predicates
Numeric values:	`integer`, `decimal`, `float`, `double`.	$=, \neq, \leq, \geq, <, >$	$=_n, \neq_n, \leq_n, \geq_n, <_n, >_n$
String values:	`string`	$=, \neq$	$=_n, \neq_n$
Boolean values:	`boolean`	$=, \neq$	$=_n, \neq_n$
Datetime values:	`dateTime`	$=, \neq, \leq, \geq, <, >$	$=_n, \neq_n, \leq_n, \geq_n, <_n, >_n$
IRI values:	IRIs	$=, \neq$	$=_n, \neq_n$

Table 4.5: The supported value transformations.

From/To	String	Float	Double	Decimal	Integer	Datetime	Boolean
String	✓	○	○	○	○	○	○
Float	✓	✓	✓	○	○	×	✓
Double	✓	✓	✓	○	○	×	✓
Decimal	✓	✓	✓	✓	✓	×	✓
Integer	✓	✓	✓	✓	✓	×	✓
Datetime	✓	×	×	×	×	✓	×
Boolean	✓	✓	✓	✓	✓	×	✓

✓ Fully supported. ○ Supported, transformation depends on the lexical value. × Not supported.

Definition 4.2 (Class Expression). Let c be a class (URI reference), R an object property expression (Definition 4.3), U a datatype property expression (Definition 4.4), D a data range, P a binary or unary predicate, and $o_1, \ldots, o_n$ individuals. A *class expression* C is a class or any complex expression between classes, properties, individuals, and data ranges that describes a set of class instances. Such an expression may involve set operations, value restrictions and cardinality constraints, and it is recursively defined as follows:

$$\begin{aligned} C \rightarrow\ & c \mid C \sqcap C \mid C \sqcup C \mid C - C \mid \exists R.C \mid \exists U.D \mid \exists R.P \mid \exists U.P \mid \exists (R,R).P \\ & \mid \exists (U,U).P \mid \geqq_{\leqq}\, n\, R \mid \geqq_{\leqq}\, n\, U \mid \geqq_{\leqq}\, n\, R.C \mid \geqq_{\leqq}\, n\, U.D \mid \{o_1, \ldots, o_n\} \end{aligned} \tag{4.3}$$

□

Definition 4.3 (Object Property Expression). Let r be an object property (URI reference), C a class expression (Definition 4.2), U a datatype property expression (Definition 4.4), and P a binary predicate. An *object property expression* R is an object property or any complex expression between properties and classes that describes a set of binary relations between class instances. Such an expression may involve set operations and value restrictions, and it is recursively defined as follows:

$$\begin{aligned} R \rightarrow\ & r \mid R \sqcap R \mid R \sqcup R \mid R - R \mid R \circ R \mid R^{-} \mid R^{+} \\ & \mid \exists (R)(R).P \mid \exists (U)(U).P \mid R \upharpoonright C \mid R \downharpoonright C \end{aligned} \tag{4.4}$$

□

Definition 4.4 (Datatype Property Expression). Let u be an datatype property (URI reference), R an object property expression (Definition 4.3), C a class expression (Definition 4.2), D a data range, and $tranf$ a value transformation function based on Table 4.5. A *datatype property expression* U is a datatype property or any complex expression between properties, classes, and data ranges that describes a set of binary relations between class instances and data values. Such an expression may involve set operations and value restrictions, and it is recursively defined as follows:

$$U \rightarrow u \mid U \sqcap U \mid U \sqcup U \mid U - U \mid R \circ U \mid U \upharpoonright C \mid U \downharpoonright D \mid trans(U) \tag{4.5}$$

□

Table 4.6: Terminological axioms used in mapping definition.

Name	Syntax	Semantics
Class inclusion	$C_1 \sqsubseteq C_2$	$C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$
Object property inclusion	$R_1 \sqsubseteq R_2$	$R_1^{\mathcal{I}} \subseteq R_2^{\mathcal{I}}$
Datatype property inclusion	$U_1 \sqsubseteq U_2$	$U_1^{\mathcal{I}} \subseteq U_2^{\mathcal{I}}$
Class equality	$C_1 \equiv C_2$	$C_1^{\mathcal{I}} = C_2^{\mathcal{I}}$
Object property equality	$R_1 \equiv R_2$	$R_1^{\mathcal{I}} = R_2^{\mathcal{I}}$
Datatype property equality	$U_1 \equiv U_2$	$U_1^{\mathcal{I}} = U_2^{\mathcal{I}}$
Individual equality	$o_1 \equiv o_2$	$o_1^{\mathcal{I}} = o_2^{\mathcal{I}}$

Below we present the mapping types supported by the SPARQL–RW mapping model and deal with class expressions, property expressions and individuals. To this end, Table 4.6 presents the terminological axioms adopted for the formal expression of mapping relationships.

Class Mapping. A class expression C_1 can be mapped to a class expression C_2 using an equivalence or subsumption relationship.

$$C_1 \equiv C_2,\ C_1 \sqsubseteq C_2,\ C_1 \sqsupseteq C_2 \tag{4.6}$$

Object Property Mapping. An object property expression R_1 can be mapped to an object property expression R_2 using an equivalence or subsumption relationship.

$$R_1 \equiv R_2,\ R_1 \sqsubseteq R_2,\ R_1 \sqsupseteq R_2 \tag{4.7}$$

Datatype Property Mapping. A datatype property expression U_1 can be mapped to a datatype property expression U_2 using an equivalence or subsumption relationship.

$$U_1 \equiv U_2,\ U_1 \sqsubseteq U_2,\ U_1 \sqsupseteq U_2 \tag{4.8}$$

Individual Mapping. An individual o_1 can mapped to an individual o_2 using an equivalence relationship.

$$o_1 \equiv o_2 \tag{4.9}$$

Remark 4.1. Mappings involving subsumption relationships ($\sqsubseteq$, $\sqsupseteq$) make the *open-world assumption*. That is, the expression on the left/right part of the mapping describes/computes instances or binary relations that are assumed to be incomplete compared to the instances or binary relations described/computed by the expression on the right/left part.

Remark 4.2. Mappings involving equivalence relationships ($\equiv$) make the *closed-world assumption*. That is, the expression on the left/right part of the mapping describes/computes instances or binary relations are assumed to be complete compared to the instances or binary relations described/computed by the expression on the right/left part.

Remark 4.3. Equivalence or subsumption relationship between property expressions, implies equivalence or subsumption between the domains and ranges of the property expressions respectively.

4.2 Supported Mapping Formalisms

As previously mentioned, in the context of data integration, a mapping model need to have the ability to support systems with different requirements in terms of *mapping expressiveness*, *map-*

ping maintenance, *query rewriting efficiency*, and *extensibility to new sources*. To this end, various mapping formalisms have been proposed, mainly focusing in supporting data integration over relational data sources. The predominant schema mapping classes in this field are the *Global-As-View*, *Local-As-View*, and *Global-and-Local-As-View*.

The SPARQL–RW mapping model, aiming to assist ontology based mediators employing different system requirements, supports all three aforementioned formalisms. Due to the fact that these schema mapping classes have been introduced in the context of integrating relational data sources and not RDF, several adaptations have been performed. The following sections explain the *Global-As-View*, *Local-As-View*, and *Global-and-Local-As-View* approaches, considering RDF as the main data model and SPARQL as the main query language.

4.2.1 Global-As-View (GAV)

The Global-As-View (GAV) [50, 29, 51, 21] approach is based on the idea that each element of the mediator ontology schema should be characterized in terms of a view (complex element expression) over the integrated data source ontologies. The mediator ontology schema is often referred to as the global schema, hence the name of the formalism.

Mapping definition in GAV requires high familiarity with all the integrated data sources, as well as deep knowledge of their schemas. However, the main advantage of this approach is its conceptual simplicity. In order to reformulate a query posed over the global ontology schema, the mediator needs to unfold the query using the views specified in the mappings. In some sense, the mapping explicitly instructs the system how to retrieve the data when evaluating the various elements of the mediator ontology schema. Generally, quality depends on how well the designer has compiled the sources into the global schema through the mapping.

Definition 4.5 (GAV Ontology Mapping). Let G be a mediator ontology schema, and let $\bar{S} = \{S_1, \ldots, S_n\}$ be n data source ontology schemas. A *Global-As-View ontology mapping* $\bar{M}$ is a set of expressions of the form $G_i(\bar{X}) \sqsupseteq Q(\bar{S})$ or $G_i(\bar{X}) \equiv Q(\bar{S})$, where:

- G_i is an ontology term (i.e., class, property, individual) in G, and appears in at most one expression in $\bar{M}$, and
- $Q(\bar{S})$ is a view (complex element expression) over the ontology terms in $\bar{S}$. □

Definition 4.6 (GAV Semantics). Let $\bar{M} = M_1, \ldots, M_l$ be a GAV ontology mapping between G and $\bar{S} = \{S_1, \ldots, S_n\}$, where M_i is of the form $G_i(\bar{X}) \sqsupseteq Q(\bar{S})$ or $G_i(\bar{X}) \equiv Q(\bar{S})$. Let g be an instance of the mediated ontology schema G and g_i be the set of tuples for the ontology term G_i in g. Let $\bar{s} = s_1, \ldots, s_n$ be instances of $S_1, \ldots, S_n$, respectively. The tuple of instances $(g, s_1, \ldots, s_n)$ is in M_R if for every $1 \leqslant i \leqslant l$, the following hold:

- If M_i is a $\equiv$ expression, then g_i is equal to the result of evaluating Q_i on $\bar{s}$.

- If M_i is a $\sqsupseteq$ expression, then g_i subsumes the result of evaluating Q_i on $\bar{s}$. □

The following example illustrates various GAV-type mappings in a data integration scenario. It specifies mappings, involving classes and properties, between the mediator ontology and the ontologies of the integrated data sources in a hypothetical data integration system. The mappings presented in this example serve also as an indicator for the expressiveness of the SPARQL–RW mapping model.

Example 4.1. Suppose the data integration scenario illustrated in Figure 4.1. The mediator ontology G describes the schema of a mediator that integrates two data sources storing information about different types of product items including books, films, and music. More specifically, the first data source preserves information about book volumes, while the other preserves information about movies and textbooks. The schemas of the integrated data sources are provided by the source ontologies S_1, S_2 respectively.

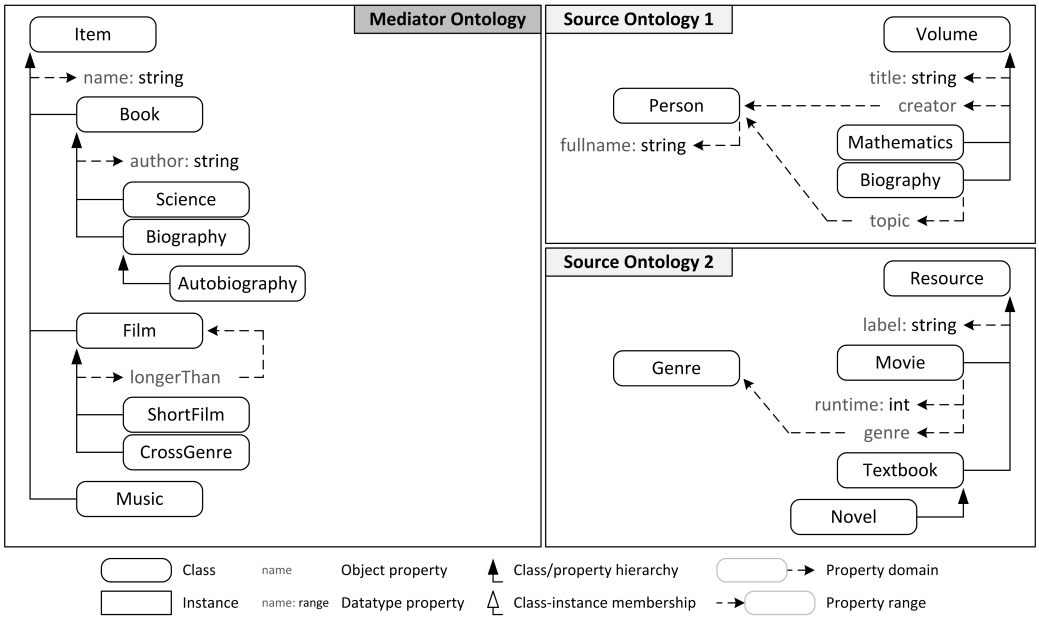


Figure 4.1: An example data integration scenario based on GAV approach.

In this kind of setting several GAV-type mappings involving classes, properties and individuals can be identified. For instance, the class `Science` of the mediator ontology can be mapped to the union of the class `Mathematics`, of the source ontology S_1, with the difference of classes `Textbook` and `Novel` of the source ontology S_2. This mapping emerges from the fact that the class `Science` seems to describe both `Mathematics` and `Textbook` individuals with the exception of `Novels`.

$$Science_G \equiv Mathematics_{S1} \sqcup (Textbook_{S2} - Novel_{S2})$$

Similarly, the mediator class `Autobiography` can be mapped to class `Biography`, of the source ontology S_1, restricted on the values of the properties `creator` and `topic`. This mapping is derived from the fact that the class `Autobiography` seems to describe `Biography` individuals having the same value for these two properties.

$$Autobiography_G \equiv Biography_{S1} \sqcap \exists(creator_{S1}, topic_{S1}). =$$

Mediator class `ShortFilm` can be mapped to the existential quantification of datatype property `runtime` of the source ontology S_2, since the class `ShortFilm` describes films that have duration less than or equal to 40.

$$ShortFilm_G \equiv \exists runtime_{S2}. \leqslant_{40}$$

Moreover, the mediator class `CrossGenre` can be mapped to the set of individuals providing more than one values for the property `genre` of the source ontology S_2. This mapping emerges from the fact that the class `CrossGenre` describes individuals of multiple genres.

$$CrossGenre_G \equiv \geqslant 2\, genre_{S2}$$

Apart from class mappings adopting the GAV formalism, several property mappings can be identified. For instance, the object property `longerThan` of the mediator ontology can be mapped to the binary relations described by applying an existential predicate restriction on the property `runtime` of the source ontology S_2. The restriction to be performed needs to include binary relations on which the individuals of the left part of the expression have greater duration compared to the individuals of the right part.

$$longerThan_G \equiv \exists(runtime_{S2})(runtime_{S2}). >$$

The datatype property `name` of the mediator ontology can be mapped to the union of the datatype properties `title`, of the source ontology S_1, and `label`, of the source ontology S_2. This mapping emerges from the fact that the binary relations described by the property `name` correspond with the binary relations described by the properties `title` and `label`.

$$name_G \sqsupseteq title_{S1} \sqcup label_{S2}$$

Similarly, the datatype property `author` of the mediator ontology can be mapped to the composition of the object property `creator` with the datatype property `fullname` of the source ontology S_1. This mapping is derived from the fact that the binary relations described by the datatype property `author` correspond with the binary relations provided by connecting the `Volume` individuals to the

`fullname` property of the class `Person`.

$$author_G \sqsupseteq creator_{S1} \circ fullname_{S1} \qquad \square$$

Although query reformulation looks easier, the GAV approach is effective whenever the data integration system is based on a set of sources that is stable. Extending the system with a new data source, may have an impact on the mapping definition of the various global ontology elements, whose associated views need to be redefined. The same holds for data source withdrawals, as well as for any changes in the schema of an integrated source. Mapping/view redefinition for any data source change comes to be added to the fact that this process, in order to be performed, requires deep schema knowledge of all the integrated data sources. As a result, a system integrating volatile data sources (in terms of their number or schema) it is unlikely to scale for a large number of sources.

4.2.2 Local-As-View (LAV)

The Local-As-View (LAV) [50, 52, 51, 21] formalism takes the opposite approach to GAV. It is based on the idea that each element of the integrated data source ontology schemas should be characterized in terms of a view (complex element expression) over the mediator ontology. In other words, instead of specifying how to compute answers of the mediator ontology schema, LAV focuses on describing each data source as precisely as possible and *independently* of any other sources. Generally, quality depends on how well the designer has characterized the sources in terms of the mediator schema.

The main advantage of LAV is the fact that data sources are described in isolation and the system (not the designer) is responsible for finding ways of combining data from multiple sources. As a result, the mapping definition task is easier (compared to GAV) and does not require from the designer deep schema knowledge of all the integrated data sources in order to specify a single mapping.

Definition 4.7 (LAV Ontology Mapping). Let G be a mediated schema, and let $\bar{S} = \{S_1, \ldots, S_n\}$ be n data source ontology schemas. A *Local-As-View ontology mapping* M is a set of expressions of the form $S_i(\bar{X}) \sqsubseteq Q_i(G)$ or $S_i(\bar{X}) \equiv Q_i(G)$, where:

- Q_i is a query (expression) over the mediated ontology G, and
- S_i is an ontology term from a data source schema in $\bar{S}$ and appears in at most one expression in M. $\square$

Definition 4.8 (LAV Semantics). Let $\bar{M} = M_1, \ldots, M_l$ be a LAV ontology mapping between G and $\bar{S} = \{S_1, \ldots, S_n\}$, where M_i is of the form $S_i(\bar{X}) \sqsubseteq Q_i(G)$ or $S_i(\bar{X}) \equiv Q_i(G)$. Let g be an instance of the mediated ontology schema G and let $\bar{s} = s_1, \ldots, s_n$ be instances of $S_1, \ldots, S_n$

respectively. The tuple of instances $(g, s_1, \ldots, s_n)$ is in M_R if for every $1 \leqslant i \leqslant l$, the following hold:

- If M_i is a $\equiv$ expression, then the result of evaluating Q_i over g is equal to s_i.
- If M_i is a $\sqsubseteq$ expression, then the result of evaluating Q_i over g subsumes s_i. □

The following example illustrates various LAV-type mappings in a data integration scenario. It specifies mappings, involving classes, properties and individuals, between the ontologies of the integrated data sources and the mediator ontology in a hypothetical data integration system. The mappings presented in this example serve also as an indicator for the expressiveness of the SPARQL–RW mapping model.

Example 4.2. Suppose the data integration scenario illustrated in Figure 4.2, introducing some minor ontology changes to the scenario presented in Figure 4.1. The mediator ontology G describes the schema of a mediator that integrates two data sources storing information about different types of product items including books, films, and music. More specifically, the first data source preserves information about book volumes, while the other preserves information about movies and textbooks. The schemas of the integrated data sources are provided by the source ontologies S_1 and S_2 respectively.

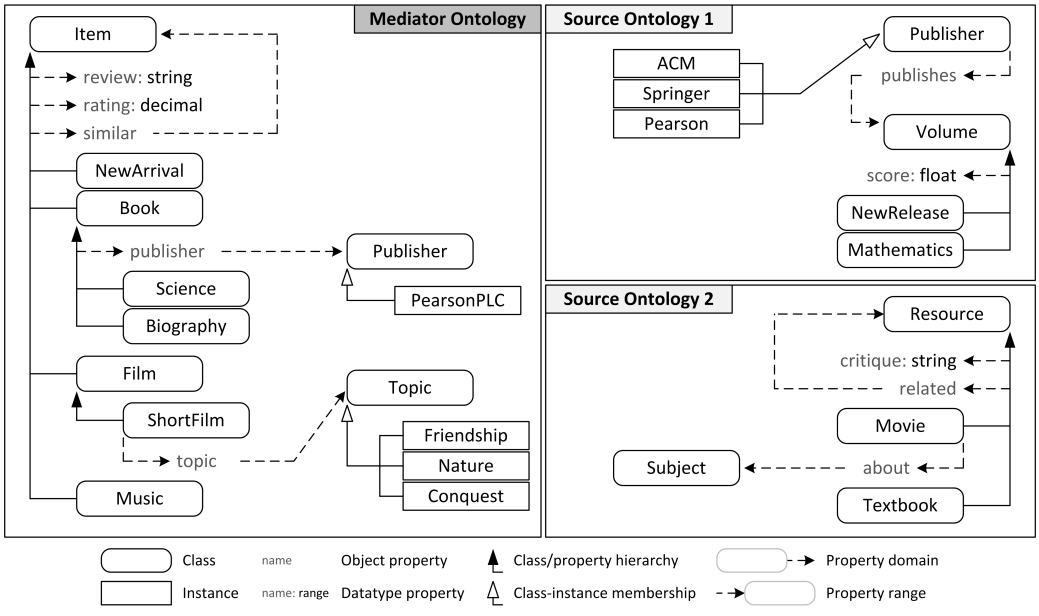


Figure 4.2: An example data integration scenario based on LAV approach.

In this kind of setting several LAV mappings involving classes, properties and individuals can be identified. For instance, the class `NewRelease` of the source ontology S_1 can be mapped to the intersection of the classes `Book` and `NewArrival` of the mediator ontology. This mapping emerges

from the fact that the class `NewRelease` seems to describe `Book` individuals which are also of type `NewArrival`.

$$NewRelease_{S1} \sqsupseteq Book_G \sqcap NewArrival_G$$

Moreover, the object property `publishes` of the source ontology S_1 can be mapped to the inverse of the object property `publisher` of the mediator ontology. This mapping emerges from the fact that the binary relations described by the object property `publishes` correspond with the inverse binary relations of property `publisher`.

$$publishes_{S1} \sqsupseteq publisher_G^-$$

The datatype property `score` of the source ontology S_1 can be mapped to the datatype property `rating` of the mediator ontology after restricting the `rating` property on its domain values and transforming its range values from decimal to float. This mapping is derived from the fact that the property `score` describes book ratings only, while the property `rating` describes ratings of various item types. Thus, the domain of property `rating` has to be restricted on `Book` individuals in order to match the domain of property `score`. Moreover, the range values of property `rating` have to be transformed from decimal to float in order to match the `score` datatype.

$$score_{S1} \sqsubseteq decimal2float(rating_G \restriction Book_G)$$

The individual `Pearson` of the source ontology S_1 can be mapped to the individual `PearsonPLC` of the mediator ontology as they seem to describe the same `Publisher`.

$$Pearson_{S1} \equiv PearsonPLC_G$$

Apart from mappings between the source ontology S_1 and the mediator ontology, several mappings between the source ontology S_2 and the mediator ontology can be identified. For instance, the class `Subject` of the source ontology S_2 can be mapped to the enumeration of individuals `Friendship`, `Nature` and `Conquest` of the mediator ontology. This mapping is derived from the fact that the class `Subject` characterizes only the aforementioned three types of movies.

$$Subject_{S2} \equiv \{Frienship_G, Nature_G, Conquest_G\}$$

Similarly, the object property `related` of the source ontology S_2 can be mapped to the transitive closure of the object property `similar` of the mediator ontology. This mapping emerges from the fact that the binary relations described by the object property `related` correspond to the binary

relations described by the transitive closure of the object property `similar`.

$$related_{S2} \sqsubseteq similar_G^+$$

Finally, the datatype property `critique` of the source ontology S_2 can be mapped to the datatype property `review` of the mediator ontology restricted on its domain property values. More specifically, the domain of the property `review` should be restricted to `Book` and `Film` individuals in order to match the domain of the property `critique`.

$$critique_{S2} \sqsubseteq review_G \restriction (Book_G \sqcup Film_G) \qquad \square$$

LAV approach is effective whenever the data integration system is based on a global ontology schema that is stable and well-established in the organization. Changes in the global ontology schema may have an impact on the mapping definition of the various source ontology elements, whose associated views need to be redefined. On the other hand, extending the system with a new source simply means the enrichment of mapping with new assertions (without performing any other changes), while data source withdrawals result simply to mapping deletion. This offers high modularity and extensibility to any data integration system adopting the LAV formalism.

Although mapping definition looks easier in LAV, query processing needs reasoning and requires the development of complex query rewriting algorithms. Finding all certain answers has been proven to be co-NP-hard in the size of the data if the queries include unions or negated predicates. The added flexibility of LAV is the reason for the increased computational complexity of answering queries. Fundamentally, the cause is that LAV enables expressing incomplete information. In contrast, the complexity of query answering in GAV is similar to that of query evaluation over a database.

4.2.3 Global-and-Local-As-View (GLAV)

The Global-and-Local-As-View (GLAV) [54, 50, 21] approach is a combination of the GAV and LAV formalisms. It offers the expressive power of both techniques, aiming to overcome their limitations. GLAV is based on the idea that a view (complex element expression) over the data source ontology schemas should be characterized in terms of a view over the mediator ontology. Query reformulation in data integration systems supporting this formalism is performed by composing the LAV techniques with the GAV techniques.

Definition 4.9 (GLAV Ontology Mapping). Let G be a mediated schema, and $\bar{S} = \{S_1, \ldots, S_n\}$ be n data source ontology schemas. A *Global-and-Local-As-View ontology mapping* M is a set of expressions of the form $Q^S(\bar{X}) \sqsubseteq Q^G(\bar{X})$ or $Q^S(\bar{X}) \equiv Q^G(\bar{X})$, where:

- Q^G is a query over the mediated ontology G whose head variables are $\bar{X}$, and

- Q^S is a query over the data sources whose head variables are also $\bar{X}$. □

Definition 4.10 (GLAV Semantics). Let $\bar{M} = M_1, \ldots, M_l$ be a GLAV ontology mapping between G and $\bar{S} = \{S_1, \ldots, S_n\}$, where M_i is of the form $Q^S(\bar{X}) \sqsubseteq Q^G(\bar{X})$ or $Q^S(\bar{X}) \equiv Q^G(\bar{X})$. Let g be an instance of the mediated ontology schema G, and let $\bar{s} = s_1, \ldots, s_n$ be instances of $S_1, \ldots, S_n$ respectively. The tuple of instances $(g, s_1, \ldots, s_n)$ is in M_R if for every $1 \leqslant i \leqslant l$, the following hold:

- If M_i is a $\equiv$ expression, then $S_i(\bar{s}) = Q_i(g)$.
- If M_i is a $\subseteq$ expression, then $S_i(\bar{s}) \sqsubseteq Q_i(g)$. □

The following example illustrates various GLAV-type mappings in a data integration scenario. It specifies mappings, involving classes and properties between the ontologies of the integrated data sources and the mediator ontology in a hypothetical data integration system. The mappings presented in this example serve also as an indicator for the expressiveness of the SPARQL–RW mapping model.

Example 4.3. Suppose the data integration scenario illustrated in Figure 4.3, introducing some minor ontology changes to the scenarios presented in the previous examples. The mediator ontology G describes the schema of a mediator integrating two data sources that store information about different types of product items including books, films, and music.

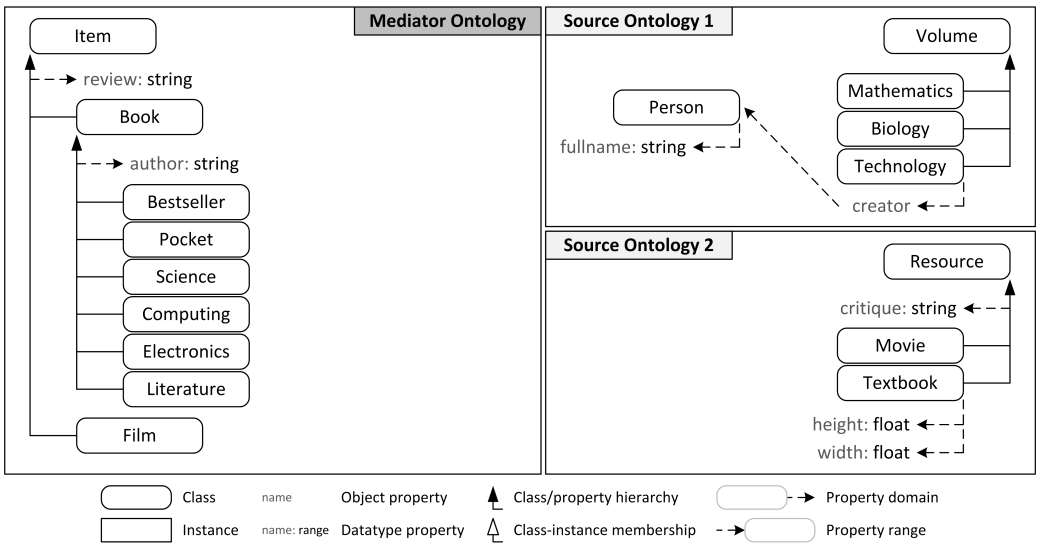


Figure 4.3: An example data integration scenario based on GLAV approach.

In this kind of setting several GLAV mappings involving classes and properties can be identified. For instance, suppose that data source S_1 is known to have only popular books regarding the subject of mathematics and biology. Therefore, the union of classes `Mathematics` and `Biology` of the

source ontology S_1 can be mapped to the intersection of classes `Science` and `Bestseller` of the mediator ontology.

$$Mathematics_{S1} \sqcup Biology_{S1} \sqsupseteq Science_G \sqcap Bestseller_G$$

Moreover, the composition of properties `creator` and `fullname`, of the source ontology S_1, can be mapped to the datatype property `author`, of the mediator ontology, restricted on its domain values. More specifically, the domain of the property `author` should be restricted to the `Computing` and `Electronics` individuals in order to match the domain of the object property `creator`.

$$creator_{S1} \circ fullname_{S1} \sqsupseteq author_G \restriction (Computing_G \sqcup Electronics_G)$$

Apart from mappings between the source ontology S_1 and the mediator ontology, several mappings between the source ontology S_2 and the mediator ontology can be identified. For instance, the datatype property `critique`, of the source ontology S_2, restricted to have `Textbook` individuals on its domain, can be mapped to the datatype property `review`, of the mediator ontology, restricted to have `Science` and `Literature` individuals on its domain.

$$critique_{S2} \restriction Textbook_{S2} \equiv review_G \restriction (Science_G \sqcup Literature_G)$$

Finally, the `Textbook` individuals of the source ontology S_2 having height and width less than 15 can be mapped to the intersection of class `Pocket`, of the mediator ontology, with the union of classes `Science` and `Literature`. This mapping emerges from the fact that the class `Pocket` of the mediator ontology describes pocket-sized books and the `Textbook` individuals of the source ontology S_2 are either literary or scientific.

$$(\exists height_{S2}. \leqslant_{15}) \sqcap (\exists width_{S2}. \leqslant_{15}) \sqsubseteq Pocket_G \sqcap (Science_G \sqcup Literature_G) \qquad \square$$

4.3 Expressive Power

The SPARQL–RW mapping model allows the specification of highly expressive ontology mappings in the context of ontology based mediator architectures. Offering constructs and axioms based on DL semantics, it enables the definition of a great variety of expressions that build upon named classes, properties and individuals of the mediator ontology and the integrated ontology schemas. It is able to support mappings between views over multiple data sources, adopting formalisms extensively used in the field of data integration.

Common constructs like *intersection*, *union*, *difference*, *enumeration*, *composition*, *inversion* and *transitivity* enable the definition of complex class and property expressions. On the other hand, *existential quantification* as well as *cardinality and existential predicate restrictions* enable

the creation of classes out of named properties. Similarly, *domain and range restrictions* are used for minimizing the binary relations described by a named property or property expression. All the aforementioned features are familiar to users dealing with the representation of specific domain knowledge and ontology management. In the context of the SPARQL–RW mapping model they are used for specifying the expressions on the left and right part of a mapping.

It is worth to mention that the SPARQL–RW mapping model does not deal with ontology mappings in general. It has been developed to support the query rewriting process in ontology based mediator architectures and therefore, the mapping capabilities of the model are highly dependent on the SPARQL expressiveness. For example, *universal quantification* is a well-known class feature in knowledge representation languages and fully supported by the OWL–DL and OWL 2 specifications. Although, this feature is extensively used for the definition of classes in the context of ontology management, it cannot be used for the definition of class mappings in the SPARQL–RW Framework since it is not exploitable by the SPARQL query rewriting process. The current specification of SPARQL misses certain aggregate functions and therefore, constructs that build upon features like universal quantification cannot be materialized.

Table 4.7 provides a feature comparison between the SPARQL–RW mapping model, the OWL-DL and OWL 2 knowledge representation languages and EDOAL. EDOAL (Expressive and Declarative Ontology Alignment Language) [20] is a highly expressive and serializable language which is used for mapping representation. It builds upon the Alignment Format [25], a well-known specification extensively used for representing alignments in ontology matching tasks.

Compared to OWL–DL mapping type capabilities, the SPARQL–RW mapping model misses features like *class complementarity* and *universal quantification*. Such constructs have not been taken into consideration since, as noted before, they cannot be exploited by the SPARQL query rewriting process. On the contrary, the SPARQL–RW mapping model offers many important features, not supported by the specification of OWL–DL, including *existential predicate* and *qualified number restrictions* for classes, *intersection*, *union*, *difference* and *composition* of properties, as well as datatype property value transformations.

On the other hand, OWL 2 has fewer differences compared to the SPARQL–RW mapping model. Unlike OWL–DL, it supports *existential predicate restrictions* and *qualified number restrictions* for classes, as well as object property *composition*. However, it still misses property *intersection*, *union* and *difference*, as well as property composition between an object and a datatype property.

Finally, as shown in Table 4.7, EDOAL is a feature full mapping representation language supporting a great variety of class and property constructs. It has been developed to describe ontology mappings in general and therefore, it does not take into account SPARQL limitations as in our case. Even by this fact, EDOAL predominates the SPARQL–RW mapping model *only* in *class complementarity*, while it does not support *instance enumeration*.

Table 4.7: Feature comparison of the SPARQL–RW mapping model, OWL–DL, OWL 2, and EDOAL.

	Basic Features	SPARQL–RW	OWL–DL	OWL 2	EDOAL
Class	Atomic (URI reference)	●	●	●	●
	Intersection	●	●	●	●
	Union	●	●	●	●
	Difference	●	○	○	●
	Complement		●	●	●
	Enumeration	●	●	●	
	Universal quantification		●	●	
	Existential quantification	●	●	●	●
	Existential predicate restriction	●		●	●
	Unqualified number restriction	●	●	●	●
	Qualified number restriction	●		●	●
	Inclusion	●	●	●	●
	Equality	●	●	●	●
Object Property	Atomic (URI reference)	●	●	●	●
	Intersection	●			●
	Union	●			●
	Difference	●			●
	Composition	●		●	●
	Inversion	●	●	●	●
	Transitive closure	●	●	●	●
	Existential predicate restriction	●			●
	Domain restriction	●	●	●	●
	Range restriction	●	●	●	●
	Inclusion	●	●	●	●
	Equality	●	●	●	●
Datatype Property	Atomic (URI reference)	●	●	●	●
	Intersection	●			●
	Union	●			●
	Difference	●			●
	Composition	●			●
	Domain restriction	●	●	●	●
	Range restriction	●	●	●	●
	Transformation	●			●
	Inclusion	●	●	●	●
	Equality	●	●	●	●
Individual	Atomic (URI reference)	●	●	●	●
	Equality	●	●	●	●

● Directly supported. ○ Indirectly supported using other constructs.

4.4 Mapping Inference

Mapping discovery is a task that can be achieved in three ways: (*a*) *manually*, defined by an expert who has a very good understanding of the ontologies to be mapped, (*b*) *automatically*, using various matching algorithms and techniques which compute similarity measures between different ontology terms, and (*c*) *semi-automatically*, using matching algorithms and techniques along with constant user feedback.

Several methods [17, 3, 27] and tools related to automatic or semi-automatic mapping discovery have been proposed and have their performance analyzed [26, 43, 90, 18, 81, 7]. Although these techniques provide satisfactory results, the quality of auto-generated mappings cannot be compared with manually specified ones. Manual mapping specification is undoubtedly a difficult process, however, it is able to provide declarative and highly expressive correspondences by exploiting domain knowledge expertise.

The SPARQL–RW mapping model does not provide any limitation regarding the mapping discovery task. To this end, mapping discovery can be performed either manually, automatically, or semi-automatically, reflecting the expressiveness of the generated mappings and subsequently the results of the query rewriting process. Supporting, however, a set of rich and flexible mappings types, it is without doubt that manual mapping definition and discovery are essential for exploiting the mapping model capabilities in their full potential. To assist the mapping discovery task, we have implemented a deductive method for performing *mapping inference*. The method exploits model-theoretic semantics and requires an initial set of mappings to be provided in order to perform effectively. It is based on reasoning over the mediator ontology schema, the integrated ontology schemas and the initial mapping set, exploiting any underlying semantics. To this end, the fact that the SPARQL–RW mapping model is based on Description Logic semantics is considered crucial.

Example 4.4 ***(Description Logic Relation Inference).*** Let S_1, S_2 be two minimal DL ontologies describing information about books. More specifically, consider that ontology S_1 contains the following statement:

$$S_1\colon\ Textbook_{S1} \equiv Mathematics_{S1} \sqcup Technology_{S1} \sqcup Biography_{S1}$$

meaning that the class `Textbook` contains exactly `Mathematics`, `Technology` and `Biography` books. In addition, consider that ontology S_2 defines book instances using the class `Book`. Having an initial mapping m expressed in DL syntax, stating that the class `Textbook` from ontology S_1 is equivalent to the class `Book` from ontology S_2:

$$m\colon\ Textbook_{S1} \equiv Book_{S2}$$

we can easily derive that the classes `Mathematics`, `Technology` and `Biography` from ontology

S_1 are subclasses of the class `Book` from ontology S_2, or even that their union is equivalent to the class `Book`.

$$Mathematics_{S1},\ Technology_{S1},\ Biography_{S1} \sqsubseteq Book_{S2}$$

$$Mathematics_{S1} \sqcup Technology_{S1} \sqcup Biography_{S1} \equiv Book_{S2}$$

The above statements are obviously entailed by the mapping m and the S_1 ontology schema information. □

Algorithm 4.1 presents the method adopted by SPARQL–RW for performing mapping inference. It takes as input the mediator ontology schema G, the ontology schemas of the integrated data sources $\bar{S}$, an initial mapping set $\bar{M}$ and the adopted formalism f in a data integration scenario. All the specified axioms in the aforementioned schemas, along with the axioms provided by the initial mapping set are merged in a new ontology schema K. After reasoning over K, the resulted inferential content is stored in a separate ontology schema R. For every unmapped ontology term in G or $\bar{S}$ (depending on the formalism f), the algorithm searches for possible mappings (equivalence or subsumption axioms) in R. The new mappings are sorted based on the relationship type and subsequently, those not specified in $\bar{M}$ and not violating the mapping formalism f are returned.

Algorithm 4.1: Mapping Inference

```
1  Function MappingInference(G, S̄, M̄, f)
       Input: A mediator ontology schema G, the integrated data source ontology schemas S̄,
              a set of mappings M̄, and the adopted formalism f.
       Output: A set of possible new relations M.
2      let K, R be empty ontology schemas;
3      let T be an empty set of ontology terms;
4      K = axioms specified in G, S̄, M̄;
5      perform reasoning over the ontology schema K;
6      R = new inferential content in K;
7      T = unmapped terms of G or S̄ (depending on f) based on M̄;
8      foreach unmapped term t ∈ T do
9          let M′ be an empty mapping set;
10         M′ = search for possible mappings of t in R;
11         sort mappings in M′ based on relationship type (order: ≡, ⊒, ⊑);
12         foreach possible mapping m ∈ M′ do
13             if m ∉ M̄ and obeys the formalism f then
14                 add the new mapping m to the mapping set M;
15             end
16         end
17     end
18     return M;
19 end
```

Mapping inference in SPARQL–RW is totally based on the specified schema axioms, on the initially provided mapping set and on reasoner capabilities. Any DL based reasoner can be used in order to perform this task, while the choice depends solely on the supported DL expressivity. It is worth to note that, currently, there is no reasoner supporting the full SPARQL–RW mapping model specification. However, most reasoners are able to support OWL 2 expressivity, and therefore, a great part of the SPARQL–RW mapping model features. Any mappings, in the initial mapping set, containing unsupported reasoner features are skipped.

As an indication for the supported expressivity of current DL based reasoners, the proposed mapping inference algorithm was implemented in SPARQL–RW using Pellet. Pellet [82] supports the full expressivity of OWL-DL ($\mathcal{SHOIN}(\mathcal{D})$) and is extended to support the OWL 2 ($\mathcal{SROIQ}(\mathcal{D})$) specification also. Apart from trivial constructs and axioms, it is able to deal with features including: *qualified cardinality restrictions*, *complex subproperty axioms*, *local reflexivity restrictions*, *reflexive, irreflexive, symmetric, and anti-symmetric properties*, *disjoint properties*, *negative property assertions*, and *user-defined dataranges*. Furthermore, Pellet provides reasoning support for inverse functional datatype properties which is an OWL Full feature.

Example 4.5 ***(Description Logic Relation Inference).*** Let S_1, S_2 be two minimal DL ontologies describing information about books. More specifically, consider that ontology S_1 contains the following statement:

$$S_1\colon\ Autobiography_{S1} \equiv Biography_{S1} \sqcap \exists(creator_{S1}, topic_{S1}). =$$

meaning that `Autobiography` books are actually `Biography` books having the same `creator` and `topic`. In addition, consider that ontology S_2 contains the following axiom:

$$S_2\colon\ Memoir_{S2} \sqsubseteq Book_{S2}$$

meaning that all instances of class `Memoir` are instances of class `Book`, but not vice versa. Having an initial mapping m expressed in DL, stating that class `Biography` from the ontology S_1 is equivalent to the class `Memoir` from the ontology S_2:

$$m\colon\ Biography_{S1} \equiv Memoir_{S2}$$

we can easily derive that the class `Autobiography` from ontology S_1 is a subclass of class `Memoir` from the ontology S_2.

$$Autobiography_{S1} \sqsubseteq Memoir_{S2}$$

The above statement is obviously entailed by the mapping m and the S_1, S_2 ontology schema information. □

Example 4.6 ***(Description Logic Relation Inference).*** Let S_1, S_2 be two minimal DL ontologies describing information about product items including books and films. More specifically, consider that ontology S_1 contains the following statement:

$$S_1\colon\ Item_{S1} \equiv Textbook_{S1} \sqcup Movie_{S1}$$

meaning that the class `Item` describes product items of type `Textbook` and `Movie`. In addition, consider that ontology S_2, apart from defining book instances using the class `Book`, contains the following axiom:

$$S_2\colon\ ShortFilm_{S2} \equiv Film_{S2} \sqcap \exists runtime_{S2}.\leqslant_{40}$$

meaning that the instances of class `ShortFilm` are exactly those of type `Film` with duration less than or equal to 40. Having two initial mappings m_1, m_2 expressed in DL syntax, stating that class `Textbook` from the ontology S_1 is equivalent to the class `Book` from the ontology S_2, and class `Movie` from the ontology S_1 is equivalent to the class `Film`:

$$m_1\colon\ Textbook_{S1} \equiv Book_{S2},\quad m_2\colon\ Movie_{S1} \equiv Film_{S2}$$

we can easily derive that the class `Item` from ontology S_1 is equivalent to the union of classes `Book` and `Film` from the ontology S_2, as well as the class `ShortFilm` from the ontology S_2 is subclass of `Movie` from the ontology S_1.

$$Item_{S1} \equiv Book_{S2} \sqcap Film_{S2},\quad ShortFilm_{S2} \sqsubseteq Movie_{S1}$$

The above statements are obviously entailed by the mappings m_1, m_2 and the S_1, S_2 ontology schema information. □

The adopted deductive method can be characterized semantics-based since model-theoretic semantics are exploited in order to compute the results. Query rewriting algorithms based on ontology mapping exploitation, need undoubtedly a mapping for every mediator ontology term in order to be able to reformulate any query posed over the mediator. Therefore, methods like the one adopted by the SPARQL–RW are considered invaluable especially when dealing with large, highly structured ontology schemas that need to be effectively integrated.

4.5 Inconsistency Identification

Mapping maintenance and correctness are extremely important tasks, especially for systems integrating volatile sources in terms of their number or schema. SPARQL–RW aiming to assist these processes, provides built-in functionality for performing *inconsistency identification* checks in terms

of *mapping formalism violations* and *semantic contradictions*. Regarding the latter, it is performed using a similar approach to mapping inference, exploiting the underlying mapping semantics.

Mapping formalism violations include multiple mapping definitions for a specific ontology term, as well as other basic errors like for example the presence of a source ontology term in the left-side expression of a GAV mapping. On the other hand, semantic contradictions include mistreated ontology terms or mappings that violate the mediator or source ontology schema axioms.

Algorithm 4.2 presents the method adopted by SPARQL–RW for identifying mapping inconsistencies. Similarly to the Algorithm 4.1, it takes as input the mediator ontology schema G, the ontology schemas of the integrated data sources $\bar{S}$, a mapping set $\bar{M}$ and the adopted formalism f in a data integration scenario. All the specified axioms in the aforementioned schemas, along with the axioms provided by the mapping set are merged in a new ontology schema K. After reasoning over K, the initial axioms of K, along with the resulted inferential content are stored in a separate ontology schema R. Every mapping in $\bar{M}$ is checked both for formalism violations and semantic contradictions in terms of R. Any mappings identified as inconsistent are returned.

Algorithm 4.2: Inconsistency Identification

```
1  Function IdentifyInconsistencies(G, S̄, M̄, f)
       Input: A mediator ontology schema G, the integrated data source ontology schemas S̄,
              a set of mappings M̄, and the adopted formalism f.
       Output: A set of mappings M identified as inconsistent in terms of the ontology
               schemas or formalism.
2      let K, R be empty ontology schemas;
3      K = axioms specified in G, S̄, M̄;
4      perform reasoning over the ontology schema K;
5      R = initial axioms of K, along with any resulted inferential content;
6      foreach mapping m ∈ M̄ do
7          if m violates the formalism f then
8              add mapping m to the mapping set M;
9          end
10         if m is inconsistent in terms of R then
11             add mapping m to the mapping set M;
12         end
13     end
14     return M;
15 end
```

Regarding semantic contradictions, they are totally based on the specified schema axioms, on the provided mapping set and on reasoner capabilities. Likewise mapping inference the majority of the DL reasoners can be used in order to perform this task. However, in this case the choice does not depend solely on the supported DL expressivity but also on the provided inference services.

Pellet [82, 44] for instance, provides services for consistency checking and concept satisfiability. Such services are important for checking the satisfiability of the provided schemas and the set of inter-schema correspondences. For any unsatisfiable term, the inter-schema correspondences should be reconsidered.

Semantic techniques are invaluable for finding mappings that lead to semantic inconsistencies. With the improvement of deductive tools, more and more systems adopt them, since they are considered a good starting base to the development of a more general approach to revision and update in mappings and networks of ontologies.

Example 4.7 ***(Mapping Formalism Violations).*** Suppose a GAV data integration scenario. The mediator ontology G describes the schema of a mediator that integrates two data sources storing information about different types of product items including books, films, and music. Let the schemas of the integrated data sources be provided by the source ontologies S_1 and S_2 respectively. In this kind of setting, consider the following set of mappings:

$$m_1 \colon \ Autobiography_G \sqsubseteq Textbook_{S2}$$

$$m_2 \colon \ Autobiography_G \equiv Biography_{S1} \sqcap \exists(creator_{S1}, topic_{S1}). =$$

The mappings m_1 and m_2 state that the mediator ontology class `Autobiography` is subclass of `Textbook` from the source ontology S_2, as well as equivalent to the class `Biography`, of the source ontology S_1, restricted on `creator` and `topic` property values. This results to a formalism violation, since GAV allows only one mapping for each mediator ontology term. Similarly, consider the following mapping:

$$m_3 \colon \ Science_G \sqcap Bestseller_G \sqsubseteq Mathematics_{S1} \sqcup Biology_{S1}$$

Mapping m_3 contains a complex expression involving multiple mediator ontology terms on the left-side expression. Taking into account that GAV allows each element of the mediator to be characterized in terms of a view over the integrated data source ontologies, mapping m_3 is considered invalid. Finally, assume the following mapping rule, stating that the class `NewRelease` of the source ontology S_1 is superclass of the intersection of mediator ontology classes `Book` and `NewArrival`:

$$m_4 \colon \ NewRelease_{S1} \sqsupseteq Book_G \sqcap NewArrival_G$$

Having specified `NewRelease` from the source ontology S_1 in the left-side expression, the GAV mapping formalism is violated and the mapping cannot be used in the query rewriting process. □

Example 4.8 ***(Semantic Contradictions).*** Let S_1, S_2 be two minimal DL ontologies describing information about product items including books and films. More specifically, consider that ontology

S_1 contains the following statement:

$$S_1\colon\ Textbook_{S1} \sqsubseteq \neg Movie_{S1}$$

meaning that all the instances of class `Textbook` are not instances of class `Movie`. In addition, consider that ontology S_2 contains the following axioms:

$$S_2\colon\ Book_{S2} \equiv Science_{S2} \sqcup Literature_{S2}$$

meaning that all instances of class `Book` are books of type `Science` or `Literature` and vice versa. Having two mappings m_1 and m_2 expressed in DL syntax, stating that class `Textbook` from ontology S_1 is equivalent to the class `Book` from ontology S_2, and the union of classes `Science` and `Literature` from ontology S_2 is subsumed by the class `Movie` from ontology S_1:

$$m_1\colon\ Textbook_{S1} \equiv Book_{S2},\quad m_2\colon\ Movie_{S1} \sqsupseteq Science_{S2} \sqcup Literature_{S2}$$

we can easily derive that the mapping m_2 is inconsistent in terms of ontology S_1, since by combining the S_1, S_2 ontology schema information along with the specified mappings m_1, m_2 we conclude that $Textbook_{S1} \sqsubseteq \neg Movie_{S1}$ and at the same time $Textbook_{S1} \sqsubseteq Movie_{S1}$. □

4.6 Mapping Language

Mapping representation is a very important issue for any system implementing a data integration scenario. Several languages have been proposed for this task including *OWL* [57], *C-OWL* [11], *SWRL* [40], MAFRA [55], the *Alignment Format* [25] and its *EDOAL* extension [20]. However, only some of them combine and satisfy core criteria including *simplicity*, *expressiveness*, *executability*, and *schema language agnosticism*. Furthermore, to the extent of our knowledge, none of the above languages considers the use of formalisms in mapping representation, while few of them are capable of describing mappings that involve views over multiple ontology schemas. A thorough comparison of these languages for the task of mapping specification in the context of data integration is available in [26].

SPARQL–RW provides a mapping representation language able to support all the mapping formalisms and features adopted by the SPARQL–RW mapping model. The general structure of the SPARQL–RW mapping language is presented in Section 4.6.1, while several mapping representation examples are provided in Section 4.6.2.

4.6.1 General Structure

In order to implement the SPARQL–RW Framework the need for a serializable language, able to support all the constructs and formalisms described by the SPARQL–RW mapping model, is of major importance. To this end, this section presents the general structure of the language adopted for mapping representation. It is based on XML syntax and the default namespace applying to the constructs in the grammar description is `http://www.music.tuc.gr/sparql-rw#`.

In the SPARQL–RW Framework, the mappings are described using the construct `model` which is presented below. Apart from the mappings, the construct contains information regarding the adopted mapping formalism, the mediator ontology (global ontology), as well as the ontologies of the integrated data sources (local ontologies). The construct `ontology` is used for describing basic ontology information including the uri, name, and description.

```
model::= <model uri="uri">
             <formalism> (GAV | LAV | GLAV) </formalism>
             <global> ontology </global>
             <locals> (ontology)+ </locals>
             <mappings> (mapping)+ </mappings>
         </model>

ontology::= <ontology uri="uri">
                (<name> string </name>)?
                (<description> string </description>)?
                (<schemaLocation> uri </schemaLocation>)?
            </ontology>
```

The mappings in the SPARQL–RW Framework consist of four basic types: (*a*) *class mappings*, mappings between class expressions specified using the `cmapping` construct, (*b*) *object property mappings*, mappings between object property expressions specified using the `opmapping` construct, (*c*) *datatype property mappings*, mappings between datatype property expressions specified using the `dpmapping` construct, and (*d*) *individual mappings*, mappings between individuals specified using the `imapping` construct. The mapping relationship in class or property mapping can be equivalence, subsumption or unspecified and is described by the construct `relation`. Regarding mappings between individuals, the only available mapping relationship is equivalence.

```
mapping::= cmapping | opmapping | dpmapping | imapping

cmapping::= <cmapping uri="uri">
                <expr1> cexpr </expr1>
```

```
                <expr2> cexpr </expr2>
                <relation> relation </relation>
            </cmapping>

opmapping::= <opmapping uri="uri">
                <expr1> opexpr </expr1>
                <expr2> opexpr </expr2>
                <relation> relation </relation>
             </opmapping>

dpmapping::= <dpmapping uri="uri">
                <expr1> dpexpr </expr1>
                <expr2> dpexpr </expr2>
                <relation> relation </relation>
             </dpmapping>

imapping::= <imapping uri="uri">
                <expr1> individual </expr1>
                <expr2> individual </expr2>
                <relation> EQUIVALENT </relation>
            </imapping>

relation::= EQUIVALENT | SUBSUMES | SUBSUMED | UNSPECIFIED
```

A class expression is specified using the construct `cexpr` and follows the abstract syntax presented in Definition 4.2. It can be either a simple class identified by its URI (`resource`) or a complex expression between classes, properties, individuals and data ranges that describes a set of class instances. Union, intersection, difference and enumeration operations are specified using the constructs `union`, `intersection`, `difference` and `enumeration`, respectively. On the other hand, existential quantification is specified by `existquant`, existential predicate restriction by `existpred` and cardinality restriction by `cardinality`.

```
cexpr::= <class> cconstruct </class>

cconstruct::= <resource uri="uri"/>
            | <union> cexpr (cexpr)+ </union>
            | <intersection> cexpr (cexpr)+ </intersection>
            | <difference> cexpr (cexpr)+ </difference>
```

```
| <enumeration> (individual)+ </enumeration>
| <existquant>
      <onOProperty> opexpr
          <quantifier> cexpr </quantifier>
      </onOProperty>
    | <onDProperty> dpexpr
          <quantifier> datatype </quantifier>
      </onDProperty>
  </existquant>
| <existpred>
      <onOProperty>
          opexpr upred
      </onOProperty>
    | <onDProperty>
          dpexpr upred
      </onDProperty>
    | <onOProperties>
          opexpr opexpr bpred
      </onOProperties>
    | <onDProperties>
          dpexpr dpexpr bpred
      </onDProperties>
  </existpred>
| <cardinality>
      <onOProperty> opexpr upred
          (<quantifier> cexpr </quantifier>)?
      </onOProperty>
    | <onDProperty> dpexpr upred
          (<quantifier> datatype </quantifier>)?
      </onDProperty>
  </cardinality>
```

On the aforementioned constructs, property constraints are introduced using the `onOProperty`, `onDProperty`, `onOProperties`, and `onDProperties` constructs, in conjunction with predicates and property quantifiers specifying the applied restrictions. To this end, binary and unary predicates are introduced using the constructs `bpred` and `upred` respectively. Both constructs are based on basic relation operators (`operator`) including $=$, $\neq$, $\leq$, $\geq$, $<$, $>$, while for the case of unary predicates an additional value, specifying the actual value restriction, is required.

```
bpred::= <predicate>
             <operator> operator </operator>
         </predicate>

upred::= <predicate>
             <operator> operator </operator>
             <value> value </value>
         </predicate>

operator::= EQUAL | NOT_EQUAL | GREATER | GREATER_EQUAL | LESS | LESS_EQUAL

value::= individual
       | <data type="base"> datavalue </data>
```

Property quantifiers are specified using the construct `quantifier` and may describe either a class expression (in case of an object property restriction) or a datatype (in case of a datatype property restriction). A datatype is introduced using the construct `datatype` and it can be either a simple XML Schema datatype (`base`) or a simple datatype restricted on a specific value condition. Value conditions are described using the construct `condition` and can be either simple or complex using the `basic`, `and`, `or`, and `not` constructs.

```
datatype::= <datatype base="base"> (condition)? </datatype>

base::= INTEGER | DECIMAL | FLOAT | DOUBLE | STRING | BOOLEAN | DATETIME

condition::= <condition> condconstruct </condition>

condconstruct::= <basic>
                     <operator> operator </operator>
                     <value> value </value>
                 </basic>
               | <and> condition (condition)+ </and>
               | <or> condition (condition)+ </or>
               | <not> condition </not>
```

An object property expression is specified using the construct `opexpr` and follows the abstract syntax presented in Definition 4.3. It can be either a simple object property identified by its URI (`resource`) or a complex expression between properties and classes that describes a set of binary relations between class instances. Union, intersection, difference and existential predicate restric-

tions are specified similarly to class expressions using the `union`, `intersection`, `difference` and `existpred` constructs. The composition operation is introduced by `composition`, inversion by `inverse`, and transitivity by the construct `transitive`. In all cases, an object property expression can be restricted on its domain and/or range values, using the construct `restrict`. Domain and range restrictions are defined using the constructs `domain` and `range` respectively, along with a class expression specifying the applied constraints.

```
opexpr::= <oproperty> opconstruct </oproperty>

opconstruct::= <resource uri="uri"/>
             | <union> opexpr (opexpr)+ </union>
             | <intersection> opexpr (opexpr)+ </intersection>
             | <difference> opexpr (opexpr)+ </difference>
             | <composition> opexpr (opexpr)+ </composition>
             | <inverse> opexpr </inverse>
             | <transitive> opexpr </transitive>
             | <existpred>
                   <onOProperties>
                       opexpr opexpr bpred
                   </onOProperties>
                 | <onDProperties>
                       dpexpr dpexpr bpred
                   </onDProperties>
               </existpred>
             | <restrict> opexpr
                   (<domain> cexpr </domain>)?
                   (<range> cexpr </range>)?
               </restrict>
```

A datatype property expression is specified using the construct `dpexpr` and follows the abstract syntax presented in Definition 4.4. It can be either a simple datatype property identified by its URI (`resource`) or a complex expression between properties, classes and data ranges that describes a set of binary relations between class instances and data values. Most datatype property operations are specified similarly to object property expressions, with the exception of range restrictions where the applied constraint is on a datatype and not on a class expression.

```
dpexpr::= <dproperty> dpconstruct </dproperty>

dpconstruct::= <resource uri="uri"/>
```

```
| <union> dpexpr (dpexpr)+ </union>
| <intersection> dpexpr (dpexpr)+ </intersection>
| <difference> dpexpr (dpexpr)+ </difference>
| <composition> (opexpr)+ dpexpr </composition>
| <transform to="base"> dpexpr </transform>
| <restrict> dpexpr
      (<domain> cexpr </domain>)?
      (<range> datatype </range>)?
  </restrict>
```

Finally, an individual is specified using the construct `individual` and it can be a simple URI.

```
individual::= <individual>
                  <resource uri="uri"/>
              </individual>
```

Appendix B describes the aforementioned structure by providing the XML Schema of the mapping language. This introduces some control to the language vocabulary and makes the SPARQL–RW Framework interoperable with external applications.

4.6.2 Mapping Examples

This section provides some mapping representation examples using the syntax described in the previous section. In order to present various mapping cases, adopting all the specified formalisms (i.e., GAV, LAV and GLAV), the mappings are based on the Examples 4.1, 4.2, and 4.3. A more comprehensive example, showing the complete set of mappings which have been specified in the Examples 4.1 is available in Appendix C.

Example 4.9. Suppose the data integration scenario illustrated in Example 4.1. The GAV mapping of the mediator class `Science`:

$$Science_G \equiv Mathematics_{S1} \sqcup (Textbook_{S2} - Novel_{S2})$$

specifying that the class `Science` is equivalent to the union of the class `Mathematics`, of the source ontology S_1, with the difference of classes `Textbook` and `Novel` of the source ontology S_2, can be described in XML syntax as follows:

```
<cmapping uri="MappingRule_a">
   <expr1>
      <class><resource uri="G:Science"/></class>
   </expr1>
```

```
   <expr2>
      <class>
         <union>
            <class><resource uri="S1:Mathematics"/></class>
            <class>
               <difference>
                  <class><resource uri="S2:Textbook"/></class>
                  <class><resource uri="S2:Novel"/></class>
               </difference>
            </class>
         </union>
      </class>
   </expr2>
   <relation>EQUIVALENT</relation>
</cmapping>
```

Similarly, the GAV mapping of the object property `longerThan` from the mediator ontology:

$$longerThan_G \equiv \exists(runtime_{S2})(runtime_{S2}). >$$

specifying that the object property `longerThan` is equivalent to the existential predicate restriction applied on the property `runtime` of the source ontology S_2 (using the binary predicate "$>$"), can be described in XML syntax as follows:

```
<opmapping uri="MappingRule_b">
   <expr1>
      <oproperty><resource uri="G:longerThan"/></oproperty>
   </expr1>
   <expr2>
      <oproperty>
         <existpred>
            <onDProperties>
               <dproperty><resource uri="S2:runtime"/></dproperty>
               <dproperty><resource uri="S2:runtime"/></dproperty>
               <predicate>
                  <operator>GREATER</operator>
               </predicate>
            </onDProperties>
         </existpred>
      </oproperty>
   </expr2>
   <relation>EQUIVALENT</relation>
</opmapping>
```

Example 4.10. Suppose the data integration scenario illustrated in Example 4.2. The LAV mapping

of class `Subject` from the source ontology S_2:

$$Subject_{S2} \equiv \{Frienship_G, Nature_G, Conquest_G\}$$

specifying that the class `Subject` is equivalent to the enumeration of the `Friendship`, `Nature` and `Conquest` class instances of the mediator ontology, can be described in XML syntax as follows:

```
<cmapping uri="MappingRule_c">
   <expr1>
      <class><resource uri="S2:Subject"/></class>
   </expr1>
   <expr2>
      <class>
         <enumeration>
            <individual><resource uri="G:Frienship"/></individual>
            <individual><resource uri="G:Nature"/></individual>
            <individual><resource uri="G:Conquest"/></individual>
         </enumeration>
      </class>
   </expr2>
   <relation>EQUIVALENT</relation>
</cmapping>
```

Similarly, the LAV mapping of the datatype property `critique` from the source ontology S_2:

$$critique_{S2} \sqsubseteq review_G \upharpoonright (Book_G \sqcup Film_G)$$

specifying that the property `critique` subsumes the datatype property `review` of the mediator ontology restricted on its domain property values (`Book` and `Film` individuals), can be described in XML syntax as follows:

```
<dpmapping uri="MappingRule_d">
   <expr1>
      <dproperty><resource uri="S2:critique"/></dproperty>
   </expr1>
   <expr2>
      <dproperty>
         <restrict>
            <dproperty><resource uri="G:review"/></dproperty>
            <domain>
               <class>
                  <union>
                     <class><resource uri="G:Book"/></class>
                     <class><resource uri="G:Film"/></class>
                  </union>
               </class>
```

```
            </domain>
         </restrict>
      </dproperty>
   </expr2>
   <relation>SUBSUMED</relation>
</dpmapping>
```

Finally, the LAV mapping of the class instance `Pearson` from the source ontology S_1:

$$Pearson_{S1} \equiv PearsonPLC_G$$

specifying that the class instance `Pearson` is equivalent to the individual `PearsonPLC` of the mediator ontology, can be described in XML syntax as follows:

```
<imapping uri="MappingRule_e">
   <expr1>
      <individual><resource uri="S1:Pearson"/></individual>
   </expr1>
   <expr2>
      <individual><resource uri="G:PearsonPLC"/></individual>
   </expr2>
   <relation>EQUIVALENT</relation>
</imapping>
```

Example 4.11. Suppose the data integration scenario illustrated in Example 4.3. The GLAV mapping of classes `Mathematics` and `Biology` from the source ontology S_1:

$$Mathematics_{S1} \sqcup Biology_{S1} \sqsupseteq Science_G \sqcap Bestseller_G$$

specifying that the union of classes `Mathematics` and `Biology` subsumes the intersection of classes `Science` and `Bestseller` of the mediator ontology, can be described in XML syntax as follows:

```
<cmapping uri="MappingRule_f">
   <expr1>
      <class>
         <union>
            <class><resource uri="S1:Mathematics"/></class>
            <class><resource uri="S1:Biology"/></class>
         </union>
      </class>
   </expr1>
   <expr2>
      <class>
         <intersection>
            <class><resource uri="G:Science"/></class>
```

```
            <class><resource uri="G:Bestseller"/></class>
         </intersection>
      </class>
   </expr2>
   <relation>SUBSUMES</relation>
</cmapping>
```

Similarly, the GLAV mapping of the properties `creator` and `fullname` of the source ontology S_1:

$$creator_{S1} \circ fullname_{S1} \sqsupseteq author_G \restriction (Computing_G \sqcup Electronics_G)$$

specifying that the composition of properties `creator` and `fullname` is equivalent to the property `author`, of the mediator ontology, restricted on its domain values (`Computing` and `Electronics` individuals), can be described in XML syntax as follows:

```
<dpmapping uri="MappingRule_g">
   <expr1>
      <dproperty>
         <composition>
            <oproperty><resource uri="S1:creator"/></oproperty>
            <dproperty><resource uri="S1:fullname"/></dproperty>
         </composition>
      </dproperty>
   </expr1>
   <expr2>
      <dproperty>
         <restrict>
            <dproperty><resource uri="G:author"/></dproperty>
            <domain>
               <class>
                  <union>
                     <class><resource uri="G:Computing"/></class>
                     <class><resource uri="G:Electronics"/></class>
                  </union>
               </class>
            </domain>
         </restrict>
      </dproperty>
   </expr2>
   <relation>SUBSUMES</relation>
</dpmapping>
```

4.7 Summary

In this chapter we presented the SPARQL–RW mapping model, a model for the expression of mappings between ontology schemas in the context of ontology based mediators. It consists of a gram-

mar defining the mapping types which can be exploited in SPARQL query rewriting, as well as a specification of the mapping type semantics. Furthermore, it is based on Description Logics and it is capable of describing a great variety of mapping types between OWL ontologies, providing *high flexibility* and *satisfying different system requirements and user needs*. The mapping model is able to support well-known mapping formalisms, including *GAV*, *LAV*, and *GLAV*, satisfying strong data integration requirements for *query rewriting efficiency* and *extensibility to new sources*. The expressiveness of the SPARQL–RW mapping model has been demonstrated by providing a feature comparison with common knowledge and mapping representation languages.

Additionally to the mapping model, we defined a mapping language based on XML syntax, being able to represent all the discussed types of inter-schema correspondences and mapping formalisms. The language combines a set of criteria including *simplicity*, *expressiveness*, *executability*, and *schema language agnosticism*. Furthermore, the use of XML Schema in mapping language definition: (*a*) provides exceptional *validation* capabilities, (*b*) supports easy mapping *serialization and deserialization*, and (*c*) enables *interoperability* with external systems and applications.

Finally, aiming to assist the mapping definition process and support the maintenance of mappings conforming to the SPARQL–RW mapping model, we provided methods for performing mapping inference and identifying inconsistencies in a given set of mappings and ontology schemas. Both methods exploit the underlying DL semantics of the SPARQL–RW mapping model and are based on the use of well-known reasoning techniques.

Chapter 5

Query Rewriting

Query rewriting is a well-known technique, extensively used for addressing various issues including *query optimization*, *query answering*, and performing *information integration*. Although, this method has been studied extensively in databases, it has received limited attention by the Semantic Web community especially for performing query mediation tasks in the context of data integration. To the extent of our knowledge, there is no system performing SPARQL 1.1 query rewriting in general, or SPARQL query rewriting by exploiting any well-known mapping formalism in the context of ontology based mediator architectures.

SPARQL–RW provides a SPARQL 1.1 query rewriting method, based on GAV ontology mappings, for performing query mediation over diverse, in terms of schema, federated RDF data sources. The proposed query rewriting algorithms are proved to provide semantics preserving queries with respect to the GAV mapping types supported by the model. The reformulated queries can be executed directly on any SPARQL federated query engine, or exploited as logical query plans by any ontology based mediator system.

Formally, let G be a global ontology schema, let $\bar{S} = \{S_1, \ldots, S_n\}$ be n data source ontology schemas, and let $\bar{M}$ be a set of mappings between G and $\bar{S}$. SPARQL–RW takes as input a SPARQL query Q_G expressed over G, and rewrites it to a semantically correspondent query $Q_{\bar{S}}$, expressed over $\bar{S}$, with respect to $\bar{M}$. Subsequently, considering a set of endpoints $\bar{E} = \{E_1, \ldots, E_m\}$, and a set of relations <$\bar{S}, \bar{E}$>, specifying the available endpoints for each integrated ontology schema, SPARQL–RW transforms the reformulated query $Q_{\bar{S}}$ to a federated one, expressed over $\bar{E}$.

This chapter presents the developed query rewriting algorithms, based on a set of inference rules and recursive DL to SPARQL transformation functions, using GAV ontology mappings supported by the model (see Chapter 4). Our method is not only useful for ontology based mediator systems, but also for any system supporting transparent query access over federated RDF data sources. Query rewriting based on LAV and GLAV approaches can be performed by combining the proposed rules and transformation functions with already existing algorithms including *Bucket* [52], *Minicon* [71,

70] and *Inverse-Rules* [24].

Section 5.1 provides an overview of the query rewriting process. Sections 5.2 and 5.3 present the functions and rules adopted for the rewriting of triple patterns that refer to data and schema respectively. Finally, Section 5.4 proposes a set of algorithms for performing SPARQL 1.1 graph pattern rewriting based on a set of predefined ontology mappings and data source endpoints.

5.1 Overview

The proposed query rewriting method is based on the reformulation of the input query graph pattern by exploiting a set of predefined ontology mappings and data source endpoints. The graph pattern is the main body of a SPARQL query and consists mainly of triple patterns, conjunctions, disjunctions, optional parts, constraints over triple pattern variables, and nested queries. To consider the importance of the graph pattern part, note that other query constructs like solution sequence modifiers and aggregates are applied on the solutions provided by matching the query graph pattern over the queried RDF dataset.

More specifically, the result of the query rewriting process is generated by replacing the graph pattern of the input query with the reformulated graph pattern, generated by the rewriting algorithm. Therefore, the method is *independent of the query type, the solution sequence modifiers and aggregates*. Furthermore, note that ontology schema terms appear only in triple patterns and infrequently in filter expressions also. Considering that our approach is based on the exploitation of ontology mappings, the provided graph pattern rewriting algorithm *depends directly on triple pattern and filter expression rewriting*. Thus, any graph operators appearing in the input query graph pattern do not affect the rewriting procedure.

The query rewriting algorithm proceeds by traversing the query execution tree in a bottom-up manner, starting by rewriting the innermost nested query graph pattern, and more specifically its triple patterns and filter expressions. Similarly the algorithm continues until the total input query has been reformulated using the predefined ontology mappings. Then, the resulted triple patterns and filter expressions of the reformulated graph pattern are grouped and enclosed in SERVICE clauses based on the appearing ontology terms and the integrated data source endpoints. The generated result is a federated SPARQL 1.1 query, referring to the integrated data source ontologies, able to be executed in any SPARQL federated query engine [2, 80, 78, 62, 76]. The rewriting process is not dependent on mapping relationships like equivalence and subsumption. Mapping relationships affect only the results of evaluating the reformulated query over the integrated RDF data sources.

Note that triple pattern rewriting is not a trivial process. Considering that a triple pattern has three parts; that is, subject-predicate-object, the provided triple pattern rewriting algorithm consists of three-steps. For any ontology term appearing in any triple pattern position, the triple pattern is reformulated using an already specified mapping for this term. The result of this process is a graph

pattern, which is subsequently reformulated triple pattern by triple pattern, in terms of the rest triple pattern parts. Any variables, blank nodes, and literal constants appearing in the input triple pattern remain intact after the rewriting process.

Unlike other query languages such as SQL and XQuery, SPARQL allows both data and schema queries. In order to do so, the language does not provide any specific construct but permits the use of triple patterns containing any RDF, RDFS, or OWL vocabulary term. Data triple patterns refer to information about instances and data values, while schema triple patterns refer to information about class/property hierarchies, property domains/ranges, etc. Having as input a set of inter-schema correspondences, data and schema triple patterns cannot be reformulated in the same way. A factor that can used in order to determine whether a triple pattern refers to data or schema is the term appearing in the triple pattern's predicate position. Since triple pattern rewriting depends on the triple pattern type, triple patterns containing a variable in the predicate position are not supported by our method.

Regarding filter expressions, the SPARQL–RW query rewriting method is able to exploit only 1:1 cardinality mappings. Variables, literal constants, operators and built-in functions appearing inside an input query's filter expression, remain intact after the rewriting process.

5.2 Data Triple Pattern Rewriting

This section presents the set of functions and rules adopted for the rewriting of data triple patterns; that is, the triple patterns referring to information about instances and data values. Considering that the rewriting of a triple pattern is a three-step procedure, involving mappings for its subject-predicate-object parts, the provided functions ensure that each step of the rewriting process preserves the exploited mapping type semantics.

Definition 5.1 (Data Triple Pattern). Let I be the set of IRIs, L the set of RDF Literals, and V the set of variables. Considering that a triple pattern is a tuple $(s, p, o) \in (I \cup V) \times (I \cup V) \times (I \cup L \cup V)$, a *data triple pattern* is a triple pattern, where:

- No variable is used in the predicate position.
- Each property used in the predicate position is either an ontology object/datatype property or one of the following built-in properties: `rdf:type`, `owl:sameAs`, `owl:differentFrom`.
- If `rdf:type` is used in the predicate position, an ontology class is used in the object position.

□

Definition 5.2 (Semantics Preserving Rewriting). Let G be a mediator ontology schema, let $\bar{S} = \{S_1, \ldots, S_n\}$ be n data source ontology schemas, and let $\bar{M}$ be a complete set of sound GAV mappings between G and $\bar{S}$. Assuming an RDF dataset DS that combines G, $\bar{S}$, $\bar{M}$, along with the respective datasets of G and $\bar{S}$, we state that *the rewriting of a triple pattern t to a graph pattern g, using a mapping $m \in \bar{M}$, is semantics preserving if and only if*:

- Given a variable set $\mathcal{J} = var(t)$, the evaluation of t and g (projected on $\mathcal{J}$) over the RDF dataset DS preserve the exploited mapping's relationship; that is:
  - If mapping m is of type equivalence ($\equiv$), then: $[[t]]_{DS} \equiv \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.
  - If mapping m is of type subsumption ($\sqsubseteq$), then: $[[t]]_{DS} \sqsubseteq \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.
  - If mapping m is of type subsumption ($\sqsupseteq$), then: $[[t]]_{DS} \sqsupseteq \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$. □

From this point forward, we adopt the notation presented in Chapter 4; that is, C for class expressions, R for object property expressions, U for datatype property expressions, o for individuals, P for unary/binary predicates and D for data ranges. Table 5.1 provides the function definitions of $\mathcal{T}_u(P, x)$, $\mathcal{T}_b(P, x, y)$ and $\mathcal{T}_d(D, x)$, that transform unary/binary predicate and data range restrictions to filter expressions. These basic transformation functions are used by the main data triple pattern rewriting methods, especially, when dealing with mappings that involve value restrictions.

Table 5.1: Predicate and data range restriction transformations. Functions $\mathcal{T}_u(P, x)$, $\mathcal{T}_b(P, x, y)$ and $\mathcal{T}_d(D, x)$ transform unary/binary predicate and data range restrictions to filter expressions.

If predicate P is of type:	**then $\mathcal{T}_u(P, x) =$**
$=_n$	$(x = n)$
$\neq_n$	$(x! = n)$
$\leq_n$	$(x <= n)$
$\geq_n$	$(x >= n)$
$<_n$	$(x < n)$
$>_n$	$(x > n)$
If predicate P is of type:	**then $\mathcal{T}_b(P, x, y) =$**
$=$	$(x = y)$
$\neq$	$(x\ ! = y)$
$\leq$	$(x <= y)$
$\geq$	$(x >= y)$
$<$	$(x < y)$
$>$	$(x > y)$
If data range D is of type:	**then $\mathcal{T}_d(D, x) =$**
dr: built-in data range	$(datatype(x) = dr)$
$dr(cond)$	$\mathcal{T}_d(dr, x)$ && $\mathcal{T}_d(cond, x)$
(P)	$\mathcal{T}_u(P, x)$
$(cond_1 \wedge cond_2)$	$\mathcal{T}_d(cond_1, x)$ && $\mathcal{T}_d(cond_2, x)$
$(cond_1 \vee cond_2)$	$\mathcal{T}_d(cond_1, x)$ || $\mathcal{T}_d(cond_2, x)$
$(!\, cond)$	$!\,\mathcal{T}_d(cond, x)$

Table 5.2 presents the function $\mathcal{D}_s(t, m)$ for the rewriting of a data triple pattern t using a mapping m for the term appearing in the subject position of t. By definition, when a class or property is used on the subject position of a triple pattern, the predicate position cannot contain an object/ datatype property, or the built-in properties `owl:sameAs` and `owl:differentFrom`. Thus, relying on the *data triple pattern* definition (Definition 5.1), the only case mentioned for the rewriting of a data triple pattern based on a mapping of its subject part concerns only individuals.

Table 5.2: Data triple pattern rewriting based on subject part. Function $\mathcal{D}_s(t, m)$ rewrites a triple pattern t using a mapping m for the term appearing in the subject position of t.

If t is of type:	**and m is of type:**	**then $\mathcal{D}_s(t, m) =$**
$(o_1, pred, ob)$	$o_1 \rightarrow o_2$	$(o_2, pred, ob)$

Similarly, Tables 5.3 and 5.4 present the functions $\mathcal{D}_p(t, m)$ and $\mathcal{D}_o(t, m)$ for the rewriting of a data triple pattern t using a mapping m for the term appearing in the predicate and object position of t, respectively. Since neither classes or individuals can be used in the predicate position of a triple pattern, only property mappings can be exploited in the rewriting of a triple pattern based on its predicate part. On the other hand, when a property is used on the object position of a triple pattern, the predicate position cannot contain an object/datatype property, or the built-in properties `rdf:type`, `owl:sameAs` and `owl:differentFrom`. Thus, relying on the *data triple pattern* definition, the only cases mentioned for the rewriting of a data triple pattern based on a mapping of its object part concern: (*a*) individuals, and (*b*) classes, with the precondition that the triple pattern's predicate position contains the `rdf:type` property.

As already mentioned, in order to fully rewrite a triple pattern, any proposed algorithm must use a mapping for each ontology term appearing in it. Thus, the functions presented in Tables 5.2, 5.3 and 5.4 can be considered as rewriting steps in the process of triple pattern rewriting. Appendix A proves that all the provided rewriting functions preserve the exploited mapping type semantics.

Lemma 5.1. Let t be a data triple pattern and m a predefined mapping for the term appearing in the subject position of t. Triple pattern t can be reformulated based on its subject part by invoking the function $\mathcal{D}_s(t, m)$, presented in Table 5.2. Considering the semantics of the initial triple pattern, as well as the semantics of the resulted graph pattern, this rewriting step is guaranteed to preserve the semantics of the exploited mapping m. The proof is available in the Appendix A.1. □

Lemma 5.2. Let t be a data triple pattern and m a predefined mapping for the term appearing in the predicate position of t. Triple pattern t can be reformulated based on its predicate part by invoking the function $\mathcal{D}_p(t, m)$, presented in Table 5.3. Considering the semantics of the initial triple pattern, as well as the semantics of the resulted graph pattern, this rewriting step is guaranteed to preserve the semantics of the exploited mapping m. The proof is available in the Appendix A.2. □

Lemma 5.3. Let t be a data triple pattern and m a predefined mapping for the term appearing in the object position of t. Triple pattern t can be reformulated based on its object part by invoking the function $\mathcal{D}_o(t, m)$, presented in Table 5.4. Considering the semantics of the initial triple pattern, as well as the semantics of the resulted graph pattern, this rewriting step is guaranteed to preserve the semantics of the exploited mapping m. The proof is available in the Appendix A.3. □

Table 5.3: Data triple pattern rewriting based on predicate part. Function $\mathcal{D}_p(t, m)$ rewrites a triple pattern t using a mapping m for the property appearing in the predicate position of t.

If t is of type:	**and m is of type:**	**then $\mathcal{D}_p(t, m) =$**
(sub, R_1, ob)	$R_1 \to r_1$	(sub, r_1, ob)
	$R_1 \to R_2 \sqcap R_3$	$\mathcal{D}_p\big(t,\ R_1 \to R_2\big)$ AND $\mathcal{D}_p\big(t,\ R_1 \to R_3\big)$
	$R_1 \to R_2 \sqcup R_3$	$\mathcal{D}_p\big(t,\ R_1 \to R_2\big)$ UNION $\mathcal{D}_p\big(t,\ R_1 \to R_3\big)$
	$R_1 \to R_2 - R_3$	$\mathcal{D}_p\big(t,\ R_1 \to R_2\big)$ MINUS $\mathcal{D}_p\big(t,\ R_1 \to R_3\big)$
	$R_1 \to R_2 \circ R_3$	$\mathcal{D}_p\big((sub, R, ?v),\ R \to R_2\big)$ AND $\mathcal{D}_p\big((?v, R, ob),\ R \to R_3\big)$
	$R_1 \to R_2^-$	$\mathcal{D}_p\big((ob, R, sub),\ R \to R_2\big)$
	$R_1 \to R_2^+$	$\mathcal{D}_p\big(t,\ R_1 \to R_2\big)$ UNION $\mathcal{D}_p\big(t,\ R_1 \to R_2 \circ R_2\big)$
	$R_1 \to \exists(R_2)(R_3).P_1$	$\mathcal{D}_p\big((sub, R, ?v_1),\ R \to R_2\big)$ AND $\mathcal{D}_p\big((ob, R, ?v_2),\ R \to R_3\big)$ FILTER $\mathcal{T}_b\big(P_1, ?v_1, ?v_2\big)$
	$R_1 \to \exists(U_1)(U_2).P_1$	$\mathcal{D}_p\big((sub, U, ?v_1),\ U \to U_1\big)$ AND $\mathcal{D}_p\big((ob, U, ?v_2),\ U \to U_2\big)$ FILTER $\mathcal{T}_b\big(P_1, ?v_1, ?v_2\big)$
	$R_1 \to R_2 \restriction C_1$	$\mathcal{D}_p\big(t,\ R_1 \to R_2\big)$ AND $\mathcal{D}_o\big((sub, rdf{:}type, C),\ C \to C_1\big)$
	$R_1 \to R_2 \downharpoonright C_1$	$\mathcal{D}_p\big(t,\ R_1 \to R_2\big)$ AND $\mathcal{D}_o\big((ob, rdf{:}type, C),\ C \to C_1\big)$
(sub, U_1, ob)	$U_1 \to u_1$	(sub, u_1, ob)
	$U_1 \to U_2 \sqcap U_3$	$\mathcal{D}_p\big(t,\ U_1 \to U_2\big)$ AND $\mathcal{D}_p\big(t,\ U_1 \to U_3\big)$
	$U_1 \to U_2 \sqcup U_3$	$\mathcal{D}_p\big(t,\ U_1 \to U_2\big)$ UNION $\mathcal{D}_p\big(t,\ U_1 \to U_3\big)$
	$U_1 \to U_2 - U_3$	$\mathcal{D}_p\big(t,\ U_1 \to U_2\big)$ MINUS $\mathcal{D}_p\big(t,\ U_1 \to U_3\big)$
	$U_1 \to R_1 \circ U_2$	$\mathcal{D}_p\big((sub, R, ?v),\ R \to R_1\big)$ AND $\mathcal{D}_p\big((?v, U, ob),\ U \to U_2\big)$
	$U_1 \to U_2 \restriction C_1$	$\mathcal{D}_p\big(t,\ U_1 \to U_2\big)$ AND $\mathcal{D}_o\big((sub, rdf{:}type, C),\ C \to C_1\big)$
	$U_1 \to U_2 \downharpoonright D_1$	$\mathcal{D}_p\big(t,\ U_1 \to U_2\big)$ FILTER $\mathcal{T}_d\big(D_1, ob\big)$
	$U_1 \to trans(U_2)$	SELECT $\big(sub,\ trans(?v)$ AS $ob\big)$ WHERE $\big(\mathcal{D}_p\big((sub, U, ?v),\ U \to U_2\big)\big)$

Note: Function $\mathcal{D}_o(t, m)$ is defined in Table 5.4 and functions $\mathcal{T}_b(P, x, y), \mathcal{T}_d(D, x)$ are defined in Table 5.1.

Table 5.4: Data triple pattern rewriting based on object part. Function $\mathcal{D}_o(t, m)$ rewrites a triple pattern t using a mapping m for the term appearing in the object position of t.

If t is of type:	**and m is of type:**	**then $\mathcal{D}_o(t, m) =$**
$(sub, rdf{:}type, C_1)$	$C_1 \to c_1$	$(sub, rdf{:}type, c_1)$
	$C_1 \to C_2 \sqcap C_3$	$\mathcal{D}_o\big(t,\ C_1 \to C_2\big)$ AND $\mathcal{D}_o\big(t,\ C_1 \to C_3\big)$
	$C_1 \to C_2 \sqcup C_3$	$\mathcal{D}_o\big(t,\ C_1 \to C_2\big)$ UNION $\mathcal{D}_o\big(t,\ C_1 \to C_3\big)$
	$C_1 \to C_2 - C_3$	$\mathcal{D}_o\big(t,\ C_1 \to C_2\big)$ MINUS $\mathcal{D}_o\big(t,\ C_1 \to C_3\big)$
	$C_1 \to \exists R_1.C_2$	$\mathcal{D}_p\big((sub, R, ?v),\ R \to R_1 \restriction C_2\big)$
	$C_1 \to \exists U_1.D_1$	$\mathcal{D}_p\big((sub, U, ?v),\ U \to U_1 \restriction D_1\big)$
	$C_1 \to \exists R_1.P_1$	$\mathcal{D}_p\big((sub, R, ?v),\ R \to R_1\big)$ FILTER $\mathcal{T}_u\big(P_1, ?v\big)$
	$C_1 \to \exists U_1.P_1$	$\mathcal{D}_p\big((sub, U, ?v),\ U \to U_1\big)$ FILTER $\mathcal{T}_u\big(P_1, ?v\big)$
	$C_1 \to \exists (R_1, R_2).P_1$	$\mathcal{D}_p\big((sub, R, ?v_1),\ R \to R_1\big)$ AND $\mathcal{D}_p\big((sub, R, ?v_2),\ R \to R_2\big)$ FILTER $\mathcal{T}_b\big(P_1, ?v_1, ?v_2\big)$
	$C_1 \to \exists (U_1, U_2).P_1$	$\mathcal{D}_p\big((sub, U, ?v_1),\ U \to U_1\big)$ AND $\mathcal{D}_p\big((sub, U, ?v_2),\ U \to U_2\big)$ FILTER $\mathcal{T}_b\big(P_1, ?v_1, ?v_2\big)$
	$C_1 \to \overset{\geqslant}{\leqslant\!\!=} n\ R_1$	SELECT DISTINCT (sub) WHERE $\big(\mathcal{D}_p\big((sub, R, ?v),\ R \to R_1\big)\big)$ GROUP BY (sub) HAVING $\big($COUNT (DISTINCT $?v)\ \overset{\geqslant}{\leqslant\!\!=} n\big)$
	$C_1 \to \overset{\geqslant}{\leqslant\!\!=} n\ U_1$	SELECT DISTINCT (sub) WHERE $\big(\mathcal{D}_p\big((sub, U, ?v),\ U \to U_1\big)\big)$ GROUP BY (sub) HAVING $\big($COUNT (DISTINCT $?v)\ \overset{\geqslant}{\leqslant\!\!=} n\big)$
	$C_1 \to \overset{\geqslant}{\leqslant\!\!=} n\ R_1.C_2$	SELECT DISTINCT (sub) WHERE $\big(\mathcal{D}_p\big((sub, R, ?v),\ R \to R_1 \restriction C_2\big)\big)$ GROUP BY (sub) HAVING $\big($COUNT (DISTINCT $?v)\ \overset{\geqslant}{\leqslant\!\!=} n\big)$
	$C_1 \to \overset{\geqslant}{\leqslant\!\!=} n\ U_1.D_1$	SELECT DISTINCT (sub) WHERE $\big(\mathcal{D}_p\big((sub, U, ?v),\ U \to U_1 \restriction D_1\big)\big)$ GROUP BY (sub) HAVING $\big($COUNT (DISTINCT $?v)\ \overset{\geqslant}{\leqslant\!\!=} n\big)$
	$C_1 \to \{o_1, \ldots, o_n\}$	VALUES (sub) $(o_1 \ldots o_n)$
$(sub, pred, o_1)$	$o_1 \to o_2$	$(sub, pred, o_2)$

Note: Function $\mathcal{D}_p(t, m)$ is defined in Table 5.3, while functions $\mathcal{T}_u(P, x)$, $\mathcal{T}_b(P, x, y)$ are defined in Table 5.1.

Example 5.1 ***(Triple Pattern Rewriting by Subject).*** Consider a triple pattern $t = (PearsonPLC,$ $owl{:}sameAs, ?x)$ and a mapping $m\colon PearsonPLC \equiv Pearson$ for the individual `PearsonPLC` appearing in the subject position of t. Based on Table 5.2 and the fact that m is of type $o_1 \rightarrow o_2$, the data triple pattern t can be reformulated by its subject part as follows:

$$\mathcal{D}_s(t, m) = (Pearson, owl{:}sameAs, ?x) \qquad \square$$

Example 5.2 ***(Triple Pattern Rewriting by Predicate).*** Consider a triple pattern $t = (?x, author,$ $?y)$ and a mapping $m\colon author \sqsupseteq creator \circ fullname$ for the datatype property `author` appearing in the predicate position of t. Based on Table 5.3 and the fact that m is of type $U_1 \rightarrow R_1 \circ U_2$, the data triple pattern t can be reformulated by its predicate part as follows:

$$\begin{aligned}\mathcal{D}_p(t, m) &= \mathcal{D}_p\big((?x, R, ?v),\ R \rightarrow creator\big) \text{ AND } \mathcal{D}_p\big((?v, U, ?y),\ U \rightarrow fullname\big) \\ &= (?x, creator, ?v) \text{ AND } (?v, fullname, ?y)\end{aligned} \qquad \square$$

Example 5.3 ***(Triple Pattern Rewriting by Predicate).*** Consider a data triple pattern $t = (?x,$ $longerThan, ?y)$ and a mapping $m\colon longerThan \equiv \exists(runtime)(runtime).{>}$ for the object property `longerThan` appearing in the predicate position of t. Based on Table 5.3 and the fact that m is of type $R_1 \rightarrow \exists(U_1)(U_2).P_1$, the data triple pattern t can be reformulated by its predicate part as follows:

$$\begin{aligned}\mathcal{D}_p(t, m) &= \mathcal{D}_p\big((?x, U, ?v_1),\ U \rightarrow runtime\big) \text{ AND } \mathcal{D}_p\big((?y, U, ?v_2),\ U \rightarrow runtime\big) \\ &\quad \text{FILTER } \mathcal{T}_b({>}, ?v_1, ?v_2) \\ &= (?x, runtime, ?v_1) \text{ AND } (?y, runtime, ?v_2) \text{ FILTER } (?v_1 {>} ?v_2)\end{aligned} \qquad \square$$

Example 5.4 ***(Triple Pattern Rewriting by Predicate).*** Consider a triple pattern $t = (?x, review,$ $?y)$ and a mapping $m\colon review \sqsupseteq critique \restriction (Movie \sqcup Textbook)$ for the datatype property `review` appearing in the predicate position of t. Based on Table 5.3 and the fact that m is of type $U_1 \rightarrow U_2 \restriction C_1$, the data triple pattern t can be reformulated by its predicate part as follows:

$$\begin{aligned}\mathcal{D}_p(t, m) &= \mathcal{D}_p(t,\ review \rightarrow critique) \text{ AND } \mathcal{D}_o\big((?x, rdf{:}type, C),\ C \rightarrow Movie \sqcup Textbook\big) \\ &= (?x, critique, ?y) \text{ AND } \big((?x, rdf{:}type, Movie) \text{ UNION } (?x, rdf{:}type, Textbook)\big)\end{aligned}$$

Note that mapping $m'\colon C \rightarrow Movie \sqcup Textbook$ is of type $C_1 \rightarrow C_2 \sqcup C_3$, and therefore $t' = (?x,$ $rdf{:}type, C)$ is reformulated by its object part as follows:

$$\begin{aligned}\mathcal{D}_o(t', m') &= \mathcal{D}_o(t',\ C \rightarrow Movie) \text{ UNION } \mathcal{D}_o(t',\ C \rightarrow Textbook) \\ &= (?x, rdf{:}type, Movie) \text{ UNION } (?x, rdf{:}type, Textbook)\end{aligned} \qquad \square$$

Example 5.5 ***(Triple Pattern Rewriting by Object).*** Consider a triple pattern $t = (?x, rdf{:}type, Science)$ and a mapping $m\colon Science \equiv Mathematics \sqcup (Textbook - Novel)$ for the class `Science` appearing in the object position of t. Based on Table 5.4 and the fact that m is of type $C_1 \rightarrow C_2 \sqcup C_3$, the data triple pattern t can be reformulated by its object part as follows:

$$\begin{aligned}\mathcal{D}_o(t, m) &= \mathcal{D}_o(t,\ Science \rightarrow Mathematics) \text{ UNION } \mathcal{D}_o(t,\ Science \rightarrow Textbook - Novel)\\ &= (?x, rdf{:}type, Mathematics) \text{ UNION } \big((?x, rdf{:}type, Textbook) \text{ MINUS}\\ &\quad (?x, rdf{:}type, Novel)\big)\end{aligned}$$

Note that mapping $m'\colon Science \rightarrow Textbook - Novel$ is of type $C_1 \rightarrow C_2 - C_3$, and therefore:

$$\begin{aligned}\mathcal{D}_o(t, m') &= \mathcal{D}_o(t,\ Science \rightarrow Textbook) \text{ MINUS } \mathcal{D}_o(t,\ Science \rightarrow Novel)\\ &= (?x, rdf{:}type, Textbook) \text{ MINUS } (?x, rdf{:}type, Novel)\end{aligned}$$

□

Example 5.6 ***(Triple Pattern Rewriting by Object).*** Consider a triple pattern $t = (?x, rdf{:}type, Autobiography)$ and a mapping $m\colon Autobiography \sqsupseteq Biography \sqcap \exists(creator, topic).=$ for the class `Autobiography` appearing in the object position of t. Based on Table 5.4 and the fact that m is of type $C_1 \rightarrow C_2 \sqcap C_3$, the data triple pattern t can be reformulated by its object part as follows:

$$\begin{aligned}\mathcal{D}_o(t, m) &= \mathcal{D}_o(t,\ Autobiography \rightarrow Biography) \text{ AND}\\ &\quad \mathcal{D}_o\big(t,\ Autobiography \rightarrow \exists(creator, topic).=\big)\\ &= (?x, rdf{:}type, Biography) \text{ AND } (?x, creator, ?v_1) \text{ AND } (?x, topic, ?v_2)\\ &\quad \text{FILTER } (?v_1 = ?v_2)\end{aligned}$$

Note that mapping $m'\colon Autobiography \rightarrow \exists(creator, topic).=$ is of type $C_1 \rightarrow \exists(R_1, R_2).P_1$, and therefore:

$$\begin{aligned}\mathcal{D}_o(t, m') &= \mathcal{D}_p\big((?x, R, ?v_1),\ R \rightarrow creator\big) \text{ AND } \mathcal{D}_p\big((?x, R, ?v_2),\ R \rightarrow topic\big)\\ &\quad \text{FILTER } \mathcal{T}_b(=, ?v_1, ?v_2)\\ &= (?x, creator, ?v_1) \text{ AND } (?x, topic, ?v_2) \text{ FILTER } (?v_1 = ?v_2)\end{aligned}$$

□

5.3 Schema Triple Pattern Rewriting

This section presents the set of functions and rules adopted for the rewriting of schema triple patterns; that is, the triple patterns referring to class/property hierarchies, property domains/ranges and other schema information. Similarly to the previous section, we adopt the notation presented in Chapter 4; that is, C for class expressions, R for object property expressions, U for datatype property expressions, o for individuals, P for unary/binary predicates and D for data ranges.

Definition 5.3 (Schema Triple Pattern). Let I be the set of IRIs, L the set of RDF Literals, and V the set of variables. Considering that a triple pattern is a tuple $(s, p, o) \in (I \cup V) \times (I \cup V) \times (I \cup L \cup V)$, a *schema triple pattern* is a triple pattern, where:

- No variable is used in the predicate position.
- Each property used in the predicate position is an RDF, RDFS, or OWL built-in property with the exception of: `owl:sameAs`, `owl:differentFrom`.
- If `rdf:type` is used in the predicate position, either a class instance is used in the subject position, or an RDF, RDFS, or OWL built-in vocabulary term is used in the object position.

□

Compared to data triple pattern rewriting, schema triple pattern rewriting introduces several difficulties in mapping exploitation since it needs to take into account the exact triple pattern semantics. Typically, data sources and ontologies provide schema information about named classes, properties and individuals. However, mappings are formed by relating unnamed expressions that describe class instances or binary relations, and it is unlikely for any data source to provide schema information about them. In order to deal with this issue, the proposed schema triple pattern rewriting method is based on a set of deductive rules and on the exploitation of any potential schema information about the named ontology terms appearing in the input mapping.

The supported schema triple patterns, able to be reformulated effectively through the use of inference, are those having on their predicate position one of the following built-in properties: `rdf:type`, `rdfs:subClassOf`, `rdfs:subPropertyOf`, `owl:equivalentClass`, `owl:equivalentProperty`, `owl:complementOf`, and `owl:disjointWith`. The rest schema triple patterns can be reformulated only in case of a given 1:1 cardinality mapping. Tables 5.5, 5.6 and 5.7 provide the deductive rules that form the basis of the proposed schema triple pattern rewriting method.

Note that due to the high expressiveness of the SPARQL–RW mapping model, an input mapping may need to be simplified in order to be used by any inference based method. The operations that determine whether a mapping should be transformed are: *enumeration*, *existential quantification*, *existential predicate restriction*, *(un)qualified number restriction*, *property inversion*, *composition*, *transitivity*, *transformation* and *property domain/range restriction*. Mappings containing such operations need to have them removed and also their mapping relationship modified respectively.

Definition 5.4 (Simplified Class Expression). Let c be a named class. A *simplified class expression* C is a class expression having any enumeration, existential quantifications, existential predicate restrictions, and (un)qualified number restrictions removed:

$$C \rightarrow c \mid C \sqcap C \mid C \sqcup C \mid C - C \tag{5.1}$$

□

Table 5.5: Deductive rules based on class axioms.

Premise No.1	**Premise No.2**	**Conclusion**	**Axiom Type**
$C_1 \equiv C_2$	$C' \sqsubseteq C_2$	$C' \sqsubseteq C_1$	Subsumption
$C_1 \sqsupseteq C_2$			
$C_1 \equiv C_2 \sqcap C_3$	$C' \sqsubseteq C_2$ and $C' \sqsubseteq C_3$		
$C_1 \sqsupseteq C_2 \sqcap C_3$			
$C_1 \equiv C_2 \sqcup C_3$	$C' \sqsubseteq C_2$ or $C' \sqsubseteq C_3$		
$C_1 \sqsupseteq C_2 \sqcup C_3$			
$C_1 \equiv C_2$	$C' \sqsupseteq C_2$	$C' \sqsupseteq C_1$	
$C_1 \sqsubseteq C_2$			
$C_1 \equiv C_2 \sqcap C_3$	$C' \sqsupseteq C_2$ or $C' \sqsupseteq C_3$		
$C_1 \sqsubseteq C_2 \sqcap C_3$			
$C_1 \equiv C_2 \sqcup C_3$	$C' \sqsupseteq C_2$ and $C' \sqsupseteq C_3$		
$C_1 \sqsubseteq C_2 \sqcup C_3$			
$C_1 \equiv C_2 - C_3$	$C' \sqsupseteq C_2$		
$C_1 \sqsubseteq C_2 - C_3$			
$C_1 \equiv C_2$	$C' \equiv C_2$	$C' \equiv C_1$	Equivalence
$C_1 \equiv C_2 \sqcap C_3$	$C' \equiv C_2$ and $C' \equiv C_3$		
$C_1 \equiv C_2 \sqcup C_3$	$C' \equiv C_2$ and $C' \equiv C_3$		
$C_1 \equiv C_2$	$C' \equiv C_2^c$	$C' \equiv C_1^c$	Complementarity
$C_1 \equiv C_2 \sqcap C_3$	$C' \equiv C_2^c$ and $C' \equiv C_3^c$		
$C_1 \equiv C_2 \sqcup C_3$	$C' \equiv C_2^c$ and $C' \equiv C_3^c$		
$C_1 \equiv C_2$	$C' \sqcap C_2 = \emptyset$	$C' \sqcap C_1 = \emptyset$	Disjointness
$C_1 \sqsubseteq C_2$			
$C_1 \equiv C_2 \sqcap C_3$	$C' \sqcap C_2 = \emptyset$ and $C' \sqcap C_3 = \emptyset$		
$C_1 \sqsubseteq C_2 \sqcap C_3$			
$C_1 \equiv C_2 \sqcup C_3$	$C' \sqcap C_2 = \emptyset$ and $C' \sqcap C_3 = \emptyset$		
$C_1 \sqsubseteq C_2 \sqcup C_3$			
$C_1 \equiv C_2 - C_3$	$C' \sqcap C_2 = \emptyset$		
$C_1 \sqsubseteq C_2 - C_3$			

Table 5.6: Deductive rules based on object property axioms.

Premise No.1	Premise No.2	Conclusion	Axiom Type
$R_1 \equiv R_2$	$R' \sqsubseteq R_2$	$R' \sqsubseteq R_1$	Subsumption
$R_1 \sqsupseteq R_2$			
$R_1 \equiv R_2 \sqcap R_3$	$R' \sqsubseteq R_2$ and $R' \sqsubseteq R_3$		
$R_1 \sqsupseteq R_2 \sqcap R_3$			
$R_1 \equiv R_2 \sqcup R_3$	$R' \sqsubseteq R_2$ or $R' \sqsubseteq R_3$		
$R_1 \sqsupseteq R_2 \sqcup R_3$			
$R_1 \equiv R_2$	$R' \sqsupseteq R_2$	$R' \sqsupseteq R_1$	
$R_1 \sqsubseteq R_2$			
$R_1 \equiv R_2 \sqcap R_3$	$R' \sqsupseteq R_2$ or $R' \sqsupseteq R_3$		
$R_1 \sqsubseteq R_2 \sqcap R_3$			
$R_1 \equiv R_2 \sqcup R_3$	$R' \sqsupseteq R_2$ and $R' \sqsupseteq R_3$		
$R_1 \sqsubseteq R_2 \sqcup R_3$			
$R_1 \equiv R_2 - R_3$	$R' \sqsupseteq R_2$		
$R_1 \sqsubseteq R_2 - R_3$			
$R_1 \equiv R_2$	$R' \equiv R_2$	$R' \equiv R_1$	Equivalence
$R_1 \equiv R_2 \sqcap R_3$	$R' \equiv R_2$ and $R' \equiv R_3$		
$R_1 \equiv R_2 \sqcup R_3$	$R' \equiv R_2$ and $R' \equiv R_3$		

Definition 5.5 (Simplified Object Property Expression). Let r be a named object property. A *simplified object property expression* R is an object property expression having any inversion, composition, transitivity and domain/range restrictions removed:

$$R \rightarrow r \mid R \sqcap R \mid R \sqcup R \mid R - R \tag{5.2}$$

□

Definition 5.6 (Simplified Datatype Property Expression). Let u be a named datatype property. A *simplified datatype property expression* U is a datatype property expression having any composition, transformation and domain/range restrictions removed:

$$U \rightarrow u \mid U \sqcap U \mid U \sqcup U \mid U - U \tag{5.3}$$

□

Tables 5.8, 5.9 and 5.10 present the relationship transformation rules applied after simplifying an input class, object property and datatype property mapping; that is, after removing any mapped expression parts that involve the aforementioned unsupported operations. Certainly, not all mapping types can be effectively transformed. For instance, any simplification on a mapping between a property and a property composition will have the composition operation removed, and therefore,

Table 5.7: Deductive rules based on datatype property axioms.

Premise No.1	Premise No.2	Conclusion	Axiom Type
$U_1 \equiv U_2$	$U' \sqsubseteq U_2$	$U' \sqsubseteq U_1$	Subsumption
$U_1 \sqsupseteq U_2$			
$U_1 \equiv U_2 \sqcap U_3$	$U' \sqsubseteq U_2$ and $U' \sqsubseteq U_3$		
$U_1 \sqsupseteq U_2 \sqcap U_3$			
$U_1 \equiv U_2 \sqcup U_3$	$U' \sqsubseteq U_2$ or $U' \sqsubseteq U_3$		
$U_1 \sqsupseteq U_2 \sqcup U_3$			
$U_1 \equiv U_2$	$U' \sqsupseteq U_2$	$U' \sqsupseteq U_1$	
$U_1 \sqsubseteq U_2$			
$U_1 \equiv U_2 \sqcap U_3$	$U' \sqsupseteq U_2$ or $U' \sqsupseteq U_3$		
$U_1 \sqsubseteq U_2 \sqcap U_3$			
$U_1 \equiv U_2 \sqcup U_3$	$U' \sqsupseteq U_2$ and $U' \sqsupseteq U_3$		
$U_1 \sqsubseteq U_2 \sqcup U_3$			
$U_1 \equiv U_2 - U_3$	$U' \sqsupseteq U_2$		
$U_1 \sqsubseteq U_2 - U_3$			
$U_1 \equiv U_2$	$U' \equiv U_2$	$U' \equiv U_1$	Equivalence
$U_1 \equiv U_2 \sqcap U_3$	$U' \equiv U_2$ and $U' \equiv U_3$		
$U_1 \equiv U_2 \sqcup U_3$	$U' \equiv U_2$ and $U' \equiv U_3$		

the entire mapped expression erased. In these cases the mapping cannot be used for the rewriting of a schema triple pattern. Note that mappings between individuals do not need to be simplified in order to be used in the schema triple pattern rewriting process.

Table 5.8: Relationship transformation rules for class mapping simplification.

Mapping Type	Condition	Transformed Mapping Type
$C_1 \equiv C_2 \sqcap C_3$	C_2 or C_3 removed/simplified	$C_1 \sqsubseteq C_2 \sqcap C_3$ or $C_1 \sqsubseteq C_2$ or $C_1 \sqsubseteq C_3$
$C_1 \equiv C_2 \sqcup C_3$	C_2 or C_3 removed/simplified	$C_1 \sqsupseteq C_2 \sqcup C_3$ or $C_1 \sqsupseteq C_2$ or $C_1 \sqsupseteq C_3$
$C_1 \equiv C_2 - C_3$	C_3 removed/simplified	$C_1 \sqsubseteq C_2 - C_3$ or $C_1 \sqsubseteq C_2$
$C_1 \sqsubseteq C_2 \sqcap C_3$	C_2 or C_3 removed/simplified	$C_1 \sqsubseteq C_2 \sqcap C_3$ or $C_1 \sqsubseteq C_2$ or $C_1 \sqsubseteq C_3$
$C_1 \sqsubseteq C_2 \sqcup C_3$	C_2 or C_3 removed/simplified	$C_1 \rightarrow C_2 \sqcup C_3$ or $C_1 \rightarrow C_2$ or $C_1 \rightarrow C_3$
$C_1 \sqsubseteq C_2 - C_3$	C_3 removed/simplified	$C_1 \sqsubseteq C_2 - C_3$ or $C_1 \sqsubseteq C_2$
$C_1 \sqsupseteq C_2 \sqcap C_3$	C_2 or C_3 removed/simplified	$C_1 \rightarrow C_2 \sqcap C_3$ or $C_1 \rightarrow C_2$ or $C_1 \rightarrow C_3$
$C_1 \sqsupseteq C_2 \sqcup C_3$	C_2 or C_3 removed/simplified	$C_1 \sqsupseteq C_2 \sqcup C_3$ or $C_1 \sqsupseteq C_2$ or $C_1 \sqsupseteq C_3$
$C_1 \sqsupseteq C_2 - C_3$	C_3 removed/simplified	$C_1 \rightarrow C_2 - C_3$ or $C_1 \rightarrow C_2$

$\rightarrow$ Unspecified relationship.

Table 5.9: Relationship transformation rules for object property mapping simplification.

Mapping Type	Condition	Transformed Mapping Type
$R_1 \equiv R_2 \sqcap R_3$	R_2 or R_3 removed/simplified	$R_1 \sqsubseteq R_2 \sqcap R_3$ or $R_1 \sqsubseteq R_2$ or $R_1 \sqsubseteq R_3$
$R_1 \equiv R_2 \sqcup R_3$	R_2 or R_3 removed/simplified	$R_1 \sqsupseteq R_2 \sqcup R_3$ or $R_1 \sqsupseteq R_2$ or $R_1 \sqsupseteq R_3$
$R_1 \equiv R_2 - R_3$	R_3 removed/simplified	$R_1 \sqsubseteq R_2 - R_3$ or $R_1 \sqsubseteq R_2$
$R_1 \equiv R_2 \upharpoonright C_1$	C_1 removed	$R_1 \sqsubseteq R_2$
$R_1 \equiv R_2 \downharpoonright C_1$	C_1 removed	$R_1 \sqsubseteq R_2$
$R_1 \sqsubseteq R_2 \sqcap R_3$	R_2 or R_3 removed/simplified	$R_1 \sqsubseteq R_2 \sqcap R_3$ or $R_1 \sqsubseteq R_2$ or $R_1 \sqsubseteq R_3$
$R_1 \sqsubseteq R_2 \sqcup R_3$	R_2 or R_3 removed/simplified	$R_1 \rightarrow R_2 \sqcup R_3$ or $R_1 \rightarrow R_2$ or $R_1 \rightarrow R_3$
$R_1 \sqsubseteq R_2 - R_3$	R_3 removed/simplified	$R_1 \sqsubseteq R_2 - R_3$ or $R_1 \sqsubseteq R_2$
$R_1 \sqsubseteq R_2 \upharpoonright C_1$	C_1 removed	$R_1 \sqsubseteq R_2$
$R_1 \sqsubseteq R_2 \downharpoonright C_1$	C_1 removed	$R_1 \sqsubseteq R_2$
$R_1 \sqsupseteq R_2 \sqcap R_3$	R_2 or R_3 removed/simplified	$R_1 \rightarrow R_2 \sqcap R_3$ or $R_1 \rightarrow R_2$ or $R_1 \rightarrow R_3$
$R_1 \sqsupseteq R_2 \sqcup R_3$	R_2 or R_3 removed/simplified	$R_1 \sqsupseteq R_2 \sqcup R_3$ or $R_1 \sqsupseteq R_2$ or $R_1 \sqsupseteq R_3$
$R_1 \sqsupseteq R_2 - R_3$	R_3 removed/simplified	$R_1 \rightarrow R_2 - R_3$ or $R_1 \rightarrow R_2$
$R_1 \sqsupseteq R_2 \upharpoonright C_1$	C_1 removed	$R_1 \rightarrow R_2$
$R_1 \sqsupseteq R_2 \downharpoonright C_1$	C_1 removed	$R_1 \rightarrow R_2$

$\rightarrow$ Unspecified relationship.

Table 5.10: Relationship transformation rules for datatype property mapping simplification.

Mapping Type	Condition	Transformed Mapping Type
$U_1 \equiv U_2 \sqcap U_3$	U_2 or U_3 removed/simplified	$U_1 \sqsubseteq U_2 \sqcap U_3$ or $U_1 \sqsubseteq U_2$ or $U_1 \sqsubseteq U_3$
$U_1 \equiv U_2 \sqcup U_3$	U_2 or U_3 removed/simplified	$U_1 \sqsupseteq U_2 \sqcup U_3$ or $U_1 \sqsupseteq U_2$ or $U_1 \sqsupseteq U_3$
$U_1 \equiv U_2 - U_3$	U_3 removed/simplified	$U_1 \sqsubseteq U_2 - U_3$ or $U_1 \sqsubseteq U_2$
$U_1 \equiv U_2 \upharpoonright C_1$	C_1 removed	$U_1 \sqsubseteq U_2$
$U_1 \equiv U_2 \downharpoonright D_1$	D_1 removed	$U_1 \sqsubseteq U_2$
$U_1 \sqsubseteq U_2 \sqcap U_3$	U_2 or U_3 removed/simplified	$U_1 \sqsubseteq U_2 \sqcap U_3$ or $U_1 \sqsubseteq U_2$ or $U_1 \sqsubseteq U_3$
$U_1 \sqsubseteq U_2 \sqcup U_3$	U_2 or U_3 removed/simplified	$U_1 \rightarrow U_2 \sqcup U_3$ or $U_1 \rightarrow U_2$ or $U_1 \rightarrow U_3$
$U_1 \sqsubseteq U_2 - U_3$	U_3 removed/simplified	$U_1 \sqsubseteq U_2 - U_3$ or $U_1 \sqsubseteq U_2$
$U_1 \sqsubseteq U_2 \upharpoonright C_1$	C_1 removed	$U_1 \sqsubseteq U_2$
$U_1 \sqsubseteq U_2 \downharpoonright D_1$	D_1 removed	$U_1 \sqsubseteq U_2$
$U_1 \sqsupseteq U_2 \sqcap U_3$	U_2 or U_3 removed/simplified	$U_1 \rightarrow U_2 \sqcap U_3$ or $U_1 \rightarrow U_2$ or $U_1 \rightarrow U_3$
$U_1 \sqsupseteq U_2 \sqcup U_3$	U_2 or U_3 removed/simplified	$U_1 \sqsupseteq U_2 \sqcup U_3$ or $U_1 \sqsupseteq U_2$ or $U_1 \sqsupseteq U_3$
$U_1 \sqsupseteq U_2 - U_3$	U_3 removed/simplified	$U_1 \rightarrow U_2 - U_3$ or $U_1 \rightarrow U_2$
$U_1 \sqsupseteq U_2 \upharpoonright C_1$	C_1 removed	$U_1 \rightarrow U_2$
$U_1 \sqsupseteq U_2 \downharpoonright D_1$	D_1 removed	$U_1 \rightarrow U_2$

$\rightarrow$ Unspecified relationship.

Example 5.7 ***(Mapping Simplification).*** Consider a class mapping $Autobiography \sqsubseteq Biography \sqcap \exists(creator, topic).=$ between the class `Autobiography` and the class `Biography` restricted on the values of the properties `creator` and `topic`. In order for this mapping to be exploitable by the schema triple pattern rewriting process, the existential predicate restriction in the right side expression has to be removed. Therefore, based on the rules presented in Table 5.8 the initial mapping is transformed to: $Autobiography \sqsubseteq Biography$. □

Example 5.8 ***(Mapping Simplification).*** Consider a property mapping $review \equiv critique \upharpoonright (Movie \sqcup Textbook)$ between the datatype property `review` and the property `critique` restricted on its domain values. In order for this mapping to be exploitable by the schema triple pattern rewriting process, the domain restriction in the right side expression has to be removed. Therefore, based on the rules presented in Table 5.10 the initial mapping is transformed to: $review \sqsubseteq critique$. □

Tables 5.11 and 5.12 present the functions $\mathcal{S}_s(t, m)$ and $\mathcal{S}_o(t, m)$ for the rewriting of a schema triple pattern t using a mapping m for the term appearing in the subject and object position of t, respectively. Unlike data triple patterns, the subject and object positions of a schema triple pattern may contain both classes and properties. Regarding schema triple pattern rewriting based on the predicate part, no functions are provided since the RDF, RDFS, and OWL properties appearing in that position do not affect the rewriting process, and thus remain intact.

Example 5.9 ***(Triple Pattern Rewriting by Subject).*** Consider a triple pattern $t = (PearsonPLC,$ $rdf{:}type, ?x)$ and a mapping $m\colon PearsonPLC \equiv Pearson$ for the individual `PearsonPLC` appearing in the subject position of t. Based on Table 5.11 and the fact that m is of type $o_1 \rightarrow o_2$, the schema triple pattern t can be reformulated by its subject part as follows:

$$\mathcal{S}_s(t, m) = (Pearson, rdf{:}type, ?x)$$ □

Example 5.10 ***(Triple Pattern Rewriting by Subject).*** Consider a triple pattern $t = (Bestseller,$ $rdfs{:}subClassOf, ?x)$, as well as a mapping $m\colon Bestseller \equiv Textbook \sqcap Popular$ for the class `Bestseller` appearing in the subject position of t. Based on Table 5.11 and the fact that m is of type $C_1 \rightarrow C_2 \sqcap C_3$, the schema triple pattern t can be reformulated by its subject part as follows:

$$\begin{aligned}\mathcal{S}_s(t, m) &= \mathcal{S}_s(t,\ Bestseller \rightarrow Textbook) \text{ UNION } \mathcal{S}_s(t,\ Bestseller \rightarrow Popular) \\ &= (Textbook, rdfs{:}subClassOf, ?x) \text{ UNION } (Popular, rdfs{:}subClassOf, ?x)\end{aligned}$$ □

Example 5.11 ***(Triple Pattern Rewriting by Subject).*** Consider a schema triple pattern $t = (name,$ $rdfs{:}subPropertyOf, ?x)$ and a mapping $m\colon name \equiv title \sqcup label$ for the datatype property `name` appearing in the subject position of t. Based on Table 5.11 and the fact that m is of type

Table 5.11: Schema triple pattern rewriting based on subject part. Function $\mathcal{S}_s(t, m)$ rewrites a triple pattern t using a mapping m for the term appearing in the subject position of t.

If t is of type:	**m is of type:**	**and t's predicate $\in$:**	**then $\mathcal{S}_s(t, m) =$**
$(C_1, pred, ob)$	$C_1 \to c_1$	*any built-in property*	$(c_1, pred, ob)$
	$C_1 \to C_2 \sqcap C_3$	`rdfs:subClassOf`	$\mathcal{S}_s\big(t,\ C_1 \to C_2\big)$ UNION $\mathcal{S}_s\big(t,\ C_1 \to C_3\big)$
		`rdf:type`, `owl:equivalentClass`, `owl:complementOf`, `owl:disjointWith`	$\mathcal{S}_s\big(t,\ C_1 \to C_2\big)$ AND $\mathcal{S}_s\big(t,\ C_1 \to C_3\big)$
	$C_1 \to C_2 \sqcup C_3$	`rdf:type`, `rdfs:subClassOf`, `owl:equivalentClass`, `owl:complementOf`, `owl:disjointWith`	
	$C_1 \to C_2 - C_3$	`rdf:type`	
		`rdfs:subClassOf`, `owl:disjointWith`	$\mathcal{S}_s\big(t,\ C_1 \to C_2\big)$
$(R_1, pred, ob)$	$R_1 \to r_1$	*any built-in property*	$(r_1, pred, ob)$
	$R_1 \to R_2 \sqcap R_3$	`rdfs:subPropertyOf`	$\mathcal{S}_s\big(t,\ R_1 \to R_2\big)$ UNION $\mathcal{S}_s\big(t,\ R_1 \to R_3\big)$
		`rdf:type`, `owl:equivalentProperty`	$\mathcal{S}_s\big(t,\ R_1 \to R_2\big)$ AND $\mathcal{S}_s\big(t,\ R_1 \to R_3\big)$
	$R_1 \to R_2 \sqcup R_3$	`rdf:type`, `rdfs:subPropertyOf`, `owl:equivalentProperty`	
	$R_1 \to R_2 - R_3$	`rdf:type`	
		`rdfs:subPropertyOf`	$\mathcal{S}_s\big(t,\ R_1 \to R_2\big)$
$(U_1, pred, ob)$	$U_1 \to u_1$	*any built-in property*	$(u_1, pred, ob)$
	$U_1 \to U_2 \sqcap U_3$	`rdfs:subPropertyOf`	$\mathcal{S}_s\big(t,\ U_1 \to U_2\big)$ UNION $\mathcal{S}_s\big(t,\ U_1 \to U_3\big)$
		`rdf:type`, `owl:equivalentProperty`	$\mathcal{S}_s\big(t,\ U_1 \to U_2\big)$ AND $\mathcal{S}_s\big(t,\ U_1 \to U_3\big)$
	$U_1 \to U_2 \sqcup U_3$	`rdf:type`, `rdfs:subPropertyOf`, `owl:equivalentProperty`	
	$U_1 \to U_2 - U_3$	`rdf:type`	
		`rdfs:subPropertyOf`	$\mathcal{S}_s\big(t,\ U_1 \to U_2\big)$
$(o_1, pred, ob)$	$o_1 \to o_2$	*any built-in property*	$(o_2, pred, ob)$

Table 5.12: Schema triple pattern rewriting based on object part. Function $\mathcal{S}_o(t, m)$ rewrites a triple pattern t using a mapping m for the term appearing in the object position of t.

If t is of type:	**m is of type:**	**and t's predicate $\in$:**	**then $\mathcal{S}_s(t, m) =$**
$(sub, pred, C_1)$	$C_1 \rightarrow c_1$	*any built-in property*	$(sub, pred, c_1)$
	$C_1 \rightarrow C_2 \sqcap C_3$	`rdf:type`, `rdfs:subClassOf`, `owl:equivalentClass`, `owl:complementOf`, `owl:disjointWith`	$\mathcal{S}_o\big(t,\ C_1 \rightarrow C_2\big)$ AND $\mathcal{S}_o\big(t,\ C_1 \rightarrow C_3\big)$
	$C_1 \rightarrow C_2 \sqcup C_3$	`rdfs:subClassOf`	$\mathcal{S}_o\big(t,\ C_1 \rightarrow C_2\big)$ UNION $\mathcal{S}_o\big(t,\ C_1 \rightarrow C_3\big)$
		`rdf:type`, `owl:equivalentClass`, `owl:complementOf`, `owl:disjointWith`	$\mathcal{S}_o\big(t,\ C_1 \rightarrow C_2\big)$ AND $\mathcal{S}_o\big(t,\ C_1 \rightarrow C_3\big)$
	$C_1 \rightarrow C_2 - C_3$	`rdf:type`	
		`owl:disjointWith`	$\mathcal{S}_o\big(t,\ C_1 \rightarrow C_2\big)$
$(sub, pred, R_1)$	$R_1 \rightarrow r_1$	*any built-in property*	$(sub, pred, r_1)$
	$R_1 \rightarrow R_2 \sqcap R_3$	`rdf:type`, `rdfs:subPropertyOf`, `owl:equivalentProperty`	$\mathcal{S}_o\big(t,\ R_1 \rightarrow R_2\big)$ AND $\mathcal{S}_o\big(t,\ R_1 \rightarrow R_3\big)$
	$R_1 \rightarrow R_2 \sqcup R_3$	`rdfs:subPropertyOf`	$\mathcal{S}_o\big(t,\ R_1 \rightarrow R_2\big)$ UNION $\mathcal{S}_o\big(t,\ R_1 \rightarrow R_3\big)$
		`rdf:type`, `owl:equivalentProperty`	$\mathcal{S}_o\big(t,\ R_1 \rightarrow R_2\big)$ AND $\mathcal{S}_o\big(t,\ R_1 \rightarrow R_3\big)$
	$R_1 \rightarrow R_2 - R_3$	`rdf:type`	
$(sub, pred, U_1)$	$U_1 \rightarrow u_1$	*any built-in property*	$(sub, pred, u_1)$
	$U_1 \rightarrow U_2 \sqcap U_3$	`rdf:type`, `rdfs:subPropertyOf`, `owl:equivalentProperty`	$\mathcal{S}_o\big(t,\ U_1 \rightarrow U_2\big)$ AND $\mathcal{S}_o\big(t,\ U_1 \rightarrow U_3\big)$
	$U_1 \rightarrow U_2 \sqcup U_3$	`rdfs:subPropertyOf`	$\mathcal{S}_o\big(t,\ U_1 \rightarrow U_2\big)$ UNION $\mathcal{S}_o\big(t,\ U_1 \rightarrow U_3\big)$
		`rdf:type`, `owl:equivalentProperty`	$\mathcal{S}_o\big(t,\ U_1 \rightarrow U_2\big)$ AND $\mathcal{S}_o\big(t,\ U_1 \rightarrow U_3\big)$
	$U_1 \rightarrow U_2 - U_3$	`rdf:type`	
$(sub, pred, o_1)$	$o_1 \rightarrow o_2$	*any built-in property*	$(sub, pred, o_2)$

$U_1 \rightarrow U_2 \sqcup U_3$, the triple pattern t can be reformulated by its subject part as follows:

$$\begin{aligned}\mathcal{S}_s(t, m) &= \mathcal{S}_s(t,\ name \rightarrow title) \text{ AND } \mathcal{S}_s(t,\ name \rightarrow label) \\ &= (title, rdfs{:}subPropertyOf, ?x) \text{ AND } (label, rdfs{:}subPropertyOf, ?x)\end{aligned}$$ □

Example 5.12 ***(Triple Pattern Rewriting by Object).*** Consider a schema triple pattern $t = (?x, rdfs{:}subClassOf, Bestseller)$ and a mapping $m\colon Bestseller \equiv Textbook \sqcap Popular$ for the class `Bestseller` appearing in the object position of t. Based on Table 5.12 and the fact that m is of type $C_1 \rightarrow C_2 \sqcap C_3$, the triple pattern t can be reformulated by its object part as follows:

$$\begin{aligned}\mathcal{S}_o(t, m) &= \mathcal{S}_o(t,\ Bestseller \rightarrow Textbook) \text{ AND } \mathcal{S}_o(t,\ Bestseller \rightarrow Popular) \\ &= (?x, rdfs{:}subClassOf, Textbook) \text{ AND } (?x, rdfs{:}subClassOf, Popular)\end{aligned}$$ □

Example 5.13 ***(Triple Pattern Rewriting by Object).*** Consider a schema triple pattern $t = (?x, rdfs{:}subPropertyOf, name)$ and a mapping $m\colon name \equiv title \sqcup label$ for the datatype property `name` appearing in the object position of t. Based on Table 5.12 and the fact that m is of type $U_1 \rightarrow U_2 \sqcup U_3$, the triple pattern t can be reformulated by its object part as follows:

$$\begin{aligned}\mathcal{S}_o(t, m) &= \mathcal{S}_o(t,\ name \rightarrow title) \text{ UNION } \mathcal{S}_o(t,\ name \rightarrow label) \\ &= (?x, rdfs{:}subPropertyOf, title) \text{ UNION } (?x, rdfs{:}subPropertyOf, label)\end{aligned}$$ □

Example 5.14 ***(Triple Pattern Rewriting by Object).*** Consider a schema triple pattern $t = (?x, owl{:}disjointWith, Science)$, as well as a mapping $m\colon Science \equiv Mathematics \sqcup (Textbook - Novel)$ for the class `Science` appearing in the object position of t. Based on Table 5.12 and the fact that m is of type $C_1 \rightarrow C_2 \sqcup C_3$, the triple pattern t can be reformulated by its object part as follows:

$$\begin{aligned}\mathcal{S}_o(t, m) &= \mathcal{S}_o(t,\ Science \rightarrow Mathematics) \text{ AND } \mathcal{S}_o(t,\ Science \rightarrow Textbook - Novel) \\ &= (?x, owl{:}disjointWith, Mathematics) \text{ AND } (?x, owl{:}disjointWith, Textbook)\end{aligned}$$

Note that mapping $m'\colon Science \rightarrow Textbook - Novel$ is of type $C_1 \rightarrow C_2 - C_3$, and therefore:

$$\mathcal{S}_o(t, m') = \mathcal{S}_o(t,\ Science \rightarrow Textbook) = (?x, owl{:}disjointWith, Textbook)$$ □

5.4 Graph Pattern Rewriting

The SPARQL–RW query rewriting method is based on the reformulation of the input query graph pattern by exploiting a set of predefined ontology mappings and data source endpoints. This section presents the set of algorithms which have been developed for performing SPARQL 1.1 graph pattern rewriting, based on a set of GAV ontology mappings. All the provided algorithms *depend directly*

on triple pattern and filter expression rewriting, and therefore, the rules and functions presented in the previous sections are extensively used. After the rewriting of a query graph pattern, the resulted triple patterns and filter expressions need to be grouped and enclosed in SERVICE clauses based on the appearing ontology terms and the integrated data source endpoints. The generated result is a federated SPARQL 1.1 query, referring to the integrated data source ontologies, able to be executed in any SPARQL federated query engine [2, 80, 78, 62, 76].

Algorithm 5.1 is the main algorithm in the process of graph pattern rewriting. It takes as input a graph pattern G, a set of GAV ontology mappings $\bar{M}$, as well as a set of relations <$\bar{S}$, $\bar{E}$> specifying the available endpoints for each integrated ontology schema. The algorithm uses recursion and traverses the input graph pattern in a bottom-up manner, starting by rewriting the innermost basic graph patterns of any nested queries and complex graph patterns (lines 24-33). Upon identifying a basic graph pattern the algorithm begins by rewriting its filter expressions (lines 3-8). For each expression it exploits 1-1 cardinality mappings to replace global ontology terms with local ones.

Following the filter expression rewriting, the algorithm proceeds with the rewriting of every triple pattern in the basic graph pattern (lines 9-22). Any triple patterns containing property paths in their predicate position are firstly transformed into graph patterns consisted exclusively of simple triple patterns; that is, triple patterns having a single property IRI in their predicate position. The transformation is performed using the function $\mathcal{P}(t)$, presented in Table 5.13, and the resulted graph patterns are subsequently reformulated using recursion (line 13). On the other hand, simple triple patterns are directly reformulated using the Algorithms 5.2 and 5.3 based on the triple pattern type.

After triple pattern rewriting, the resulted graph pattern replaces the initial triple pattern (line 21) and the algorithm continues until the total input graph pattern G has been reformulated. The final graph pattern is passed to Algorithm 5.4 in order to be decomposed with the use of SERVICE clauses. Then, the decomposed graph pattern is ready to replace the input query graph pattern, and transform the query into a federated one referring to the integrated data sources.

Table 5.13: Property path transformations. Function $\mathcal{P}(t)$ transforms a triple pattern t containing a property path in its predicate position, into an equivalent graph pattern consisted of simple triple patterns.

If t of type:	**then $\mathcal{P}(t) =$**
(sub, iri, ob)	(sub, iri, ob)
$(sub, !iri, ob)$	unsupported
$(sub, \hat{}path, ob)$	$\mathcal{P}\big((ob, path, sub)\big)$
$(sub, path_1/path_2, ob)$	$\mathcal{P}\big((sub, path_1, ?v)\big)$ AND $\mathcal{P}\big((?v, path_2, ob)\big)$
$(sub, path_1\|path_2, ob)$	$\mathcal{P}\big((sub, path_1, ob)\big)$ UNION $\mathcal{P}\big((sub, path_2, ob)\big)$
$(sub, path*, ob)$	$\mathcal{P}\big((sub, path, ob)\big)$
$(sub, path+, ob)$	
$(sub, path?, ob)$	
$\big(sub, (path), ob\big)$	$\mathcal{P}\big((sub, path, ob)\big)$

Algorithm 5.1: Graph Pattern Rewriting

```
1  Function GraphPatternRW(G, M̄, <S̄, Ē>)
       Input: A graph pattern G, a set of mappings M̄, as well as a set of relations <S̄, Ē>
              specifying the available endpoints for each integrated ontology schema.
       Output: The reformulated graph pattern G.
2      if G is basic graph pattern then
           /* rewrite filter expressions                                    */
3          foreach filter expression f ∈ G do
4              foreach ontology term o ∈ f do
5                  let m ∈ M̄ be a 1-1 cardinality mapping for o;
6                  use m to replace o with the mapped ontology term;
7              end
8          end
           /* rewrite triple patterns                                       */
9          foreach triple pattern t ∈ G do
10             let G′ be an empty graph pattern;
11             if t contains property path then
12                 G′ = P(t) ;                          /* refer to Table 5.13 */
13                 G′ = GraphPatternRW(g, M̄);
14             else
15                 if t is data triple pattern then
16                     G′ = DataTripleRW(G, M̄);
17                 else
18                     G′ = SchemaTripleRW(G, M̄);
19                 end
20             end
21             replace t with the reformulated graph pattern G′;
22         end
23     else
24         let G′ be an empty graph pattern;
25         foreach sub-query q ∈ G do
26             let g be the graph pattern of q;
27             G′ = GraphPatternRW(g, M̄);
28             replace the graph pattern of q with G′;
29         end
30         foreach graph pattern g ∈ G do
31             G′ = GraphPatternRW(g, M̄);
32             replace g with the reformulated graph pattern G′;
33         end
34     end
35     G = GraphPatternDecomposition(G, <S̄, Ē>);
36     return G;
37 end
```

Algorithm 5.2: Data Triple Pattern Rewriting

```
1  Function DataTripleRW(G, M̄)
       Input: A graph pattern G, and a set of mappings M̄.
       Output: The reformulated graph pattern G.
2      let V, B, L, I_b be the sets of variables, blank nodes, literals and built-in IRIs;
3      if G is basic graph pattern then
4          foreach triple pattern t ∈ G do
5              let G′ be an empty graph pattern;
               /* triple pattern rewriting by predicate                    */
6              if predicate(t) ∉ {I_b} and ∃m ∈ M̄ for predicate(t) then
7                  G′ = D_p(t, m) ;                    /* refer to Table 5.3 */
8                  G′ = DataTripleRW(G′, M̄);
               /* triple pattern rewriting by subject                      */
9              else if subject(t) ∉ {V, B} and ∃m ∈ M̄ for subject(t) then
10                 G′ = D_s(t, m) ;                    /* refer to Table 5.2 */
11                 G′ = DataTripleRW(G′, M̄);
               /* triple pattern rewriting by object                       */
12             else if object(t) ∉ {V, B, L} and ∃m ∈ M̄ for object(t) then
13                 G′ = D_o(t, m) ;                    /* refer to Table 5.4 */
14             else
15                 G′ = t;
16             end
17             replace t with the reformulated graph pattern G′;
18         end
19     else
20         let G′ be an empty graph pattern;
           /* rewrite existing sub-queries                                 */
21         foreach sub-query q ∈ G do
22             let g be the graph pattern of q;
23             G′ = DataTripleRW(g, M̄);
24             replace the graph pattern of q with G′;
25         end
           /* rewrite existing sub-graph patterns                          */
26         foreach graph pattern g ∈ G do
27             G′ = DataTripleRW(g, M̄);
28             replace g with the reformulated graph pattern G′;
29         end
30     end
31     return G;
32 end
```

Algorithm 5.2 performs data triple pattern rewriting on a given graph pattern, using the rules and functions presented in Section 5.2. It takes as input a graph pattern G containing data triple patterns, as well as a set of GAV ontology mappings $\bar{M}$. The algorithm uses recursion and traverses the input graph pattern in a bottom-up manner, starting by rewriting the innermost basic graph patterns of any possible nested queries and complex graph patterns (lines 20-29). Every identified basic graph pattern is reformulated triple pattern by triple pattern using mappings for any ontology terms appearing in the subject, predicate and object triple pattern positions (lines 4-18).

Algorithm 5.3: Schema Triple Pattern Rewriting

```
1  Function SchemaTripleRW(G, M̄)
       Input: A graph pattern G, and a set of mappings M̄.
       Output: The reformulated graph pattern G.
2      let V, B be the sets of variables and blank nodes;
3      if G is basic graph pattern then
4          foreach triple pattern t ∈ G do
5              let G′ be an empty graph pattern;
               /* triple pattern rewriting by subject                    */
6              if subject(t) ∉ {V, B} and ∃m ∈ M̄ for subject(t) then
7                  m = simplify(m);
8                  G′ = S_s(t, m) ;                  /* refer to Table 5.11 */
9                  G′ = SchemaTripleRW(G′, M̄);
               /* triple pattern rewriting by object                     */
10             else if object(t) ∉ {V, B} and ∃m ∈ M̄ for object(t) then
11                 m = simplify(m);
12                 G′ = S_o(t, m) ;                  /* refer to Table 5.12 */
13             else
14                 G′ = t;
15             end
16             replace t with the reformulated graph pattern G′;
17         end
18     else
19         let G′ be an empty graph pattern;
           /* rewrite existing sub-graph patterns                        */
20         foreach graph pattern g ∈ G do
21             G′ = SchemaTripleRW(g, M̄);
22             replace g with the reformulated graph pattern G′;
23         end
24     end
25     return G;
26 end
```

Algorithm 5.4: Graph Pattern Decomposition

```
1  Function GraphPatternDecomposition(G, <S̄, Ē>)
       Input: A graph pattern G, as well as a set of relations <S̄, Ē> specifying the available
              endpoints for each integrated ontology schema.
       Output: The transformed graph pattern G.
2      if G is basic graph pattern then
3          let G′ be an empty group graph pattern;
4          G′ = use <S̄, Ē> to organize the triple patterns and filter expressions of G into
           element groups that have to be executed over the same endpoints, based on the origin
           (ontology schema) of the appearing ontology terms;
5          foreach graph pattern g ∈ G′ do
6              let E be the set of endpoints where g has to be executed;
7              if |E| = 0 then
8                  E = all the integrated data source endpoints;
9              end
10             if |E| > 1 then
11                 let g′ be an empty union graph pattern;
12                 foreach endpoint e ∈ E do
13                     add a copy of g to g′ and enclose it in a SERVICE clause;
14                     specify e as the endpoint of the SERVICE clause;
15                 end
16                 replace g with g′;
17             else
18                 enclose g in a SERVICE clause;
19                 specify the only element of E as the SERVICE clause endpoint;
20             end
21         end
22         replace G with the transformed graph pattern G′;
23     else
24         let G′ be an empty graph pattern;
25         foreach sub-query q ∈ G do
26             let g be the graph pattern of q;
27             G′ = GraphPatternDecomposition(g, <S̄, Ē>);
28             replace the graph pattern of q with G′;
29         end
30         foreach graph pattern g ∈ G do
31             G′ = GraphPatternDecomposition(g, <S̄, Ē>);
32             replace g with the transformed graph pattern G′;
33         end
34     end
35     return G;
36 end
```

Note that data triple pattern rewriting based on the predicate part and on property mappings involving inversions, results in the swapping of the subject and object parts of the input triple pattern. Therefore, for not skipping the rewriting of any global ontology term, the algorithm starts triple pattern rewriting by exploiting a mapping for the triple pattern's predicate part (line 7). The result of this process is a graph pattern, which is subsequently recursively reformulated in terms of the rest triple pattern parts (line 8). The resulted graph pattern replaces the initial triple pattern (line 17) and the algorithm continues until the total input graph pattern G has been reformulated.

On the other hand, Algorithm 5.3 performs schema triple pattern rewriting on a given graph pattern, using the rules and functions presented in Section 5.3. It takes as input a graph pattern G containing schema triple patterns, as well as a set of GAV ontology mappings $\bar{M}$. Similarly to Algorithm 5.2, it uses recursion and traverses the input graph pattern in a bottom-up manner, starting by rewriting the innermost basic graph patterns of any nested graph patterns (lines 19-23). Every identified basic graph pattern is reformulated triple pattern by triple pattern using mappings for any ontology terms appearing in the subject and object triple pattern positions (lines 4-18). Note that before being used the mappings are firstly simplified (lines 7 and 11) based on the Definitions 5.4, 5.5, 5.6, and the transformation rules of Tables 5.8, 5.9, 5.10.

The algorithm starts the rewriting of a triple pattern based on a mapping of its subject part (line 8). The result of this process is a graph pattern, which is subsequently recursively reformulated in terms of the object triple pattern part (line 9). The resulted graph pattern replaces the initial triple pattern (line 16) and the algorithm continues until the total input graph pattern G has been reformulated. Note that in both Algorithms 5.2 and 5.3, the variables and blank nodes appearing in the input graph pattern are being preserved by the rewriting process.

Algorithm 5.4 is responsible for the decomposition of the reformulated graph pattern with the use of SERVICE clauses indicating the SPARQL endpoint where each graph pattern part is to be executed. It takes as input an already reformulated graph pattern G, as well as a set of relations $<\bar{S}, \bar{E}>$ specifying the available endpoints for each integrated ontology schema. Similarly to the previously presented algorithms, Algorithm 5.4 uses recursion and traverses the input graph pattern in a bottom-up manner, starting by the innermost basic graph patterns of any possible nested queries and complex graph patterns (lines 24-33). Upon identifying a basic graph pattern the algorithm proceeds by organizing its triple patterns and filter expressions into element groups that have to be executed over the same endpoints, based on the origin (ontology schema) of the appearing ontology terms. Subsequently, the groups are enclosed in SERVICE clauses based on the endpoints where they have to be executed. Note that for triple patterns whose origin cannot be identified, the adopted strategy is to execute them over all the integrated data source endpoints. The resulted graph pattern replaces the initial one (line 22) and the algorithm continues until the total input graph pattern G has been decomposed.

The examples that follow illustrate the query rewriting process in detail. To this end, consider the data integration scenario presented in Figure 5.1. The mediator ontology G describes the schema of a mediator that integrates two data sources storing information about different types of product items including books, films, and music. More specifically, the first data source preserves information about book volumes, while the other preserves information about movies and textbooks. The schemas of the integrated data sources are provided by the source ontologies S_1 and S_2, while their SPARQL endpoints are considered to be E_1 and E_2 respectively.

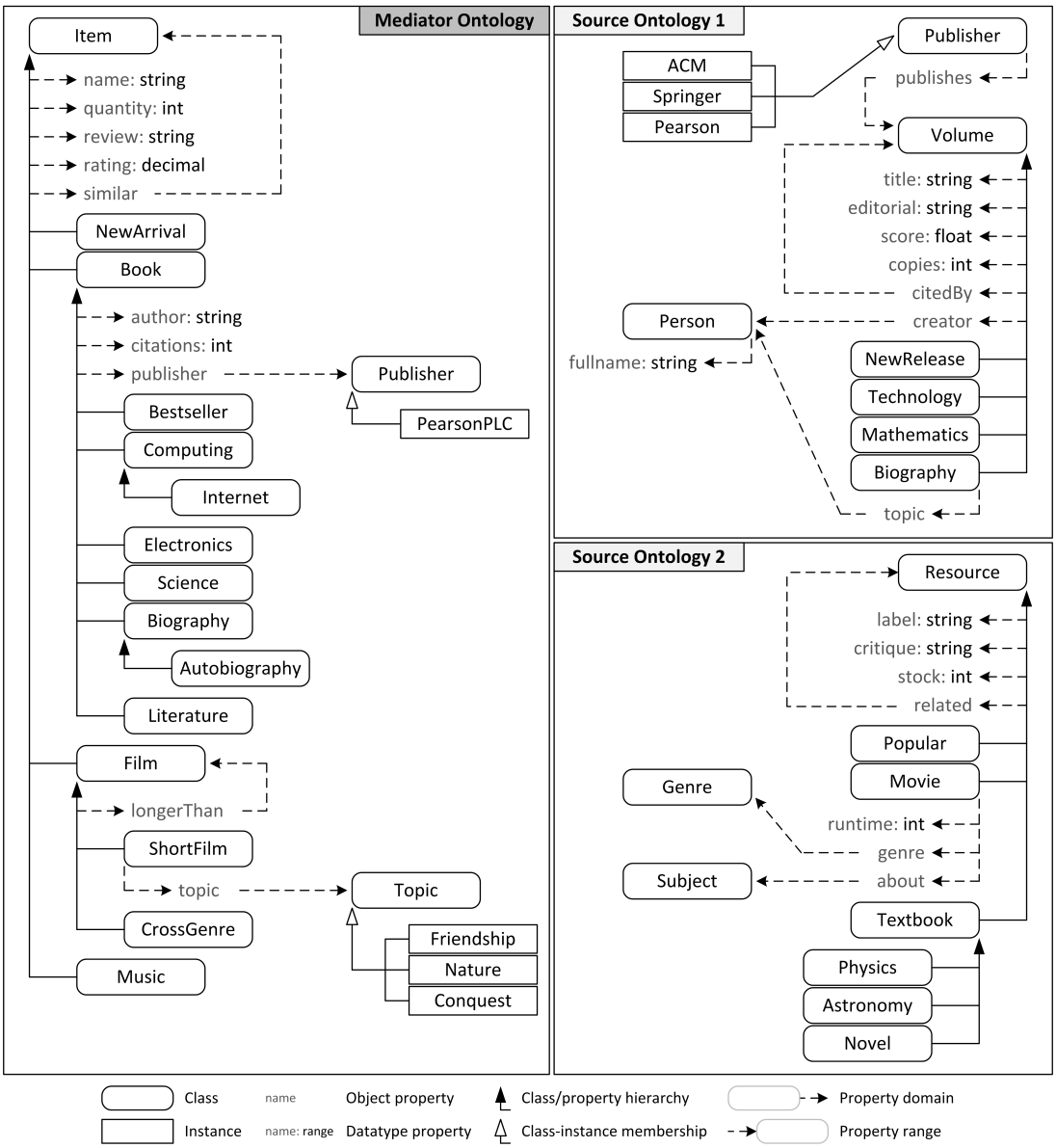


Figure 5.1: A full data integration scenario based on GAV approach.

Example 5.15. Consider the following query expressed in terms of the mediator ontology of Figure 5.1: "Return book titles written by `Jeffrey D. Ullman`, along with their publisher.". The SPARQL syntax of this query is depicted below.

```
@PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@PREFIX G:   <http://example.org/mediator#>

SELECT DISTINCT ?title ?publisher
WHERE {
   ?item G:name      ?title .
   ?item rdf:type    G:Book .
   ?item G:author    ?author .
   ?item G:publisher ?publisher .
   FILTER (?author = "Jeffrey D. Ullman")
}
```

Additionally, let $\bar{M}$ be a set of predefined GAV ontology mappings between the mediator ontology and the integrated data sources S_1 and S_2, consisted of the following mappings:

$$m_1 \colon name_G \sqsupseteq title_{S1} \sqcup label_{S2}$$
$$m_2 \colon Book_G \sqsupseteq Volume_{S1} \sqcup Textbook_{S2}$$
$$m_3 \colon author_G \sqsupseteq creator_{S1} \circ fullname_{S1}$$
$$m_4 \colon publisher_G \sqsupseteq publishes^{-}_{S1}$$

In order to rewrite the input query for being executed over the integrated data sources, it suffices to reformulate the query graph pattern using the mapping set $\bar{M}$ and the Algorithm 5.1. Apart from the initial query graph pattern and the mapping set $\bar{M}$, Algorithm 5.1 takes as input the ontology schemas $\bar{S} = \{S_1, S_2\}$ and the SPARQL endpoints $\bar{E} = \{E_1, E_2\}$ of the integrated data sources. Since the input query graph pattern consists basically of a basic graph pattern, the algorithm proceeds directly with the rewriting of its filter expressions and triple patterns.

Regarding the filter expression, it remains unchanged since it does not contain any global ontology term. On the other hand, Figure 5.2 presents an overview of the triple pattern rewriting process, skipping reformulation based on the subject part since the input triple patterns contain variables in this position. Note that the triple patterns of the input query graph pattern refer to data and therefore, the Algorithm 5.2 is used for their reformulation.

After the rewriting of filter expressions and triple patterns, the resulted graph pattern has to be decomposed, with the use of SERVICE clauses, based on the origin (ontology schema) of the appearing ontology terms. To this end, the algorithm passes the resulted graph pattern, along with the ontology schemas $\bar{S}$ and the SPARQL endpoints $\bar{E}$ to the Algorithm 5.4. The decomposed graph pattern substitutes the initial query graph pattern and the resulted query is depicted below.

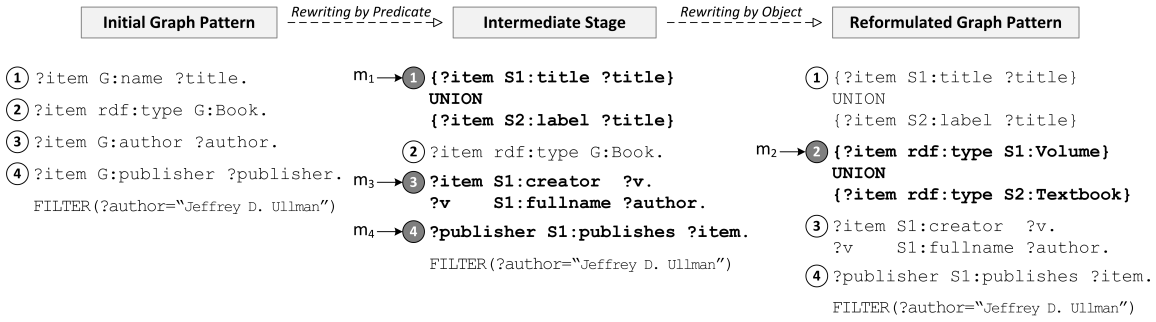


Figure 5.2: Triple pattern rewriting in the query graph pattern of Example 5.15. The parameters in the left side of the arrows denote the mappings exploited by the rewriting process.

```
@PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@PREFIX S1:  <http://example.org/source1#>
@PREFIX S2:  <http://example.org/source2#>

SELECT DISTINCT ?title ?publisher
WHERE {
   {SERVICE <E1> {?item S1:title ?title}}
   UNION
   {SERVICE <E2> {?item S2:label ?title}}

   {SERVICE <E1> {?item rdf:type S1:Volume}}
   UNION
   {SERVICE <E2> {?item rdf:type S2:Textbook}}

   SERVICE <E1> {
      ?item S1:creator  ?v .
      ?v    S1:fullname ?author .
      ?publisher S1:publishes ?item .
      FILTER (?author = "Jeffrey D. Ullman")
   }
}
```

Example 5.16. Consider the following query expressed in terms of the mediator ontology of Figure 5.1: "Return bestselling scientific or autobiography book titles.". The SPARQL syntax of this query is depicted below.

```
@PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@PREFIX G:   <http://example.org/mediator#>

SELECT DISTINCT ?title
WHERE {
   ?item G:name   ?title .
   ?item rdf:type G:Bestseller .
   {?item rdf:type G:Science}
   UNION
```

```
    {?item rdf:type G:Autobiography}
}
```

Additionally, let $\bar{M}$ be a set of predefined GAV ontology mappings between the mediator ontology and the integrated data sources S_1 and S_2, consisted of the following mappings:

$$m_1 \colon name_G \sqsupseteq title_{S1} \sqcup label_{S2}$$
$$m_2 \colon Bestseller_G \sqsubseteq Popular_{S2}$$
$$m_3 \colon Science_G \equiv Mathematics_{S1} \sqcup (Textbook_{S2} - Novel_{S2})$$
$$m_4 \colon Autobiography_G \equiv Biography_{S1} \sqcap \exists(creator_{S1}, topic_{S1}). =$$

Similarly to the previous example, in order to rewrite the input query for being executed over the integrated data sources, it suffices to reformulate the query graph pattern using the mapping set $\bar{M}$ and the Algorithm 5.1. In this case, the input query graph pattern consists of two triple patterns and a union graph pattern. The algorithm proceeds in a bottom-up approach and rewrites any triple patterns in the input graph pattern. Figure 5.3 presents an overview of the triple pattern rewriting process, skipping reformulation based on the subject part since the input triple patterns contain variables in this position. Note that the triple patterns of the input query graph pattern refer to data and therefore, the Algorithm 5.2 is used for their reformulation.

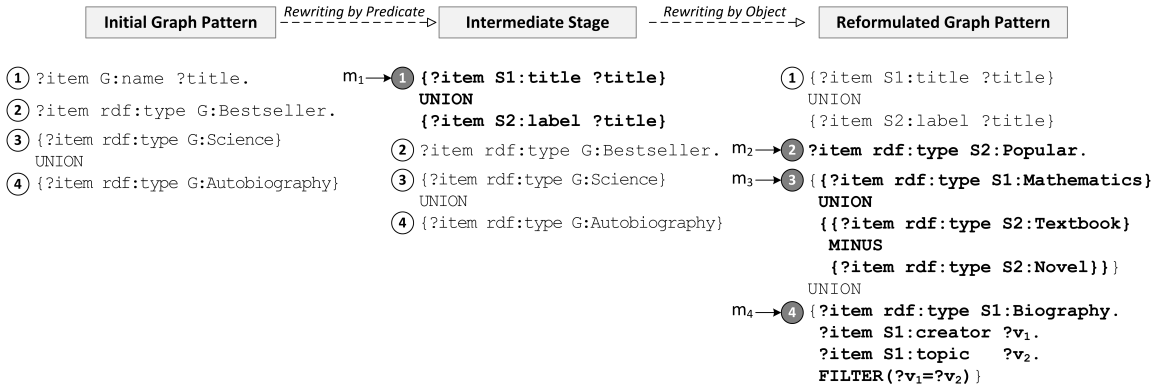


Figure 5.3: Triple pattern rewriting in the query graph pattern of Example 5.16. The parameters in the left side of the arrows denote the mappings exploited by the rewriting process.

After triple pattern rewriting, the resulted graph pattern is decomposed using the Algorithm 5.4, based on the origin (ontology schema) of the appearing ontology terms. The final query is formed by substituting the initial query graph pattern with the decomposed graph pattern. Considering E_1 and E_2 as the SPARQL endpoints of the integrated data sources S_1 and S_2 respectively, the resulted query is depicted below.

```
@PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@PREFIX S1:  <http://example.org/source1#>
@PREFIX S2:  <http://example.org/source2#>

SELECT DISTINCT ?title
WHERE {
   {SERVICE <E1> {?item S1:title ?title}}
   UNION
   {SERVICE <E2> {?item S2:label ?title}}

   SERVICE <E2> {?item rdf:type S2:Popular}

   {
      {SERVICE <E1> {?item rdf:type S1:Mathematics}}
      UNION
      {{SERVICE <E2> {?item rdf:type S2:Textbook}}
       MINUS
       {SERVICE <E2> {?item rdf:type S2:Novel}}}
   }
   UNION
   {
      SERVICE <E1> {
         ?item rdf:type   S1:Biography .
         ?item S1:creator ?v1 .
         ?item S1:topic   ?v2 .
         FILTER (?v1 = ?v2)
      }
   }
}
```

Example 5.17. Consider the following query expressed in terms of the mediator ontology of Figure 5.1: "Return at most 10 cross-genre short films. The results should be formed in descending order based on their title values". The SPARQL syntax of this query is depicted below.

```
@PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@PREFIX G:   <http://example.org/mediator#>

SELECT DISTINCT ?title
WHERE {
   ?item G:name   ?title .
   ?item rdf:type G:CrossGenre .
   ?item rdf:type G:ShortFilm .
} ORDER BY DESC (?title) LIMIT 10
```

Additionally, let $\bar{M}$ be a set of predefined GAV ontology mappings between the mediator ontology

and the integrated data sources S_1 and S_2, consisted of the following mappings:

$$m_1 \colon name_G \sqsupseteq title_{S1} \sqcup label_{S2}$$
$$m_2 \colon CrossGenre_G \equiv \geqslant 2\, genre_{S2}$$
$$m_3 \colon ShortFilm_G \equiv \exists runtime_{S2}.\, \leqslant_{40}$$

Similarly to the previous examples, in order to rewrite the input query for being executed over the integrated data sources, it suffices to reformulate the query graph pattern using the mapping set $\bar{M}$ and the Algorithm 5.1. Since the input query graph pattern consists basically of a basic graph pattern, the algorithm proceeds directly with the rewriting of its triple patterns. Figure 5.4 presents an overview of the triple pattern rewriting process, skipping reformulation based on the subject part since the input triple patterns contain variables in this position. Note that the triple patterns of the input query graph pattern refer to data and therefore, the Algorithm 5.2 is used for their reformulation.

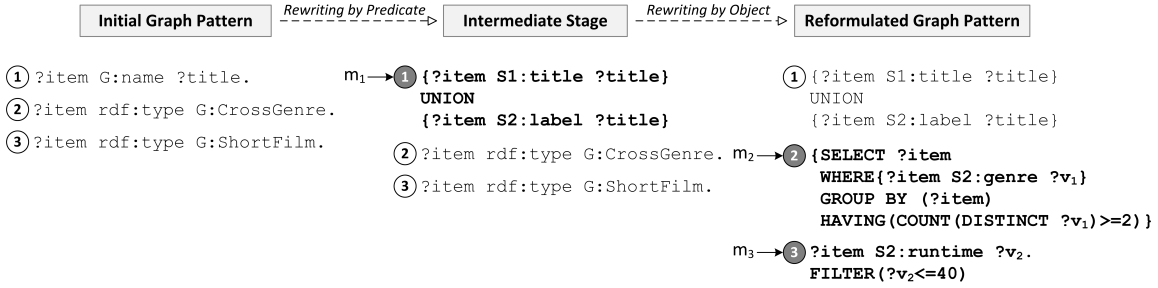


Figure 5.4: Triple pattern rewriting in the query graph pattern of Example 5.17. The parameters in the left side of the arrows denote the mappings exploited by the rewriting process.

After triple pattern rewriting, the resulted graph pattern is decomposed using the Algorithm 5.4, based on the origin (ontology schema) of the appearing ontology terms. The final query is formed by substituting the initial query graph pattern with the decomposed graph pattern. Considering E_1 and E_2 as the SPARQL endpoints of the integrated data sources S_1 and S_2 respectively, the resulted query is depicted below.

```
@PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@PREFIX S1:  <http://example.org/source1#>
@PREFIX S2:  <http://example.org/source2#>

SELECT DISTINCT ?title
WHERE {
   {SERVICE <E1> {?item S1:title ?title}}
   UNION
   {SERVICE <E2> {?item S2:label ?title}}
```

```
    {SELECT ?item
     WHERE {SERVICE <E2> {?item S2:genre ?v1}}
     GROUP BY (?item)
     HAVING (COUNT(DISTINCT ?v1) >= 2)}
    SERVICE <E2> {
       ?item S2:runtime  ?v2 .
       FILTER (?v2 <= 40)
    }
} ORDER BY DESC (?title) LIMIT 10
```

5.5 Summary

In this chapter we presented the SPARQL–RW query rewriting method, a formal method for performing query mediation over diverse, in terms of schema, federated RDF data sources. It *considers the SPARQL 1.1 specification and it is able to handle both data and schema queries*. The proposed query rewriting strategy is based on the reformulation of the input query graph pattern and it is *independent of the query type, the solution sequence modifiers and aggregates*.

The presented query rewriting algorithms *rely on triple pattern and filter expression rewriting* using a set of predefined GAV ontology mappings between the mediator ontology schema and the integrated data source ontology schemas. Triple pattern rewriting is *based on a complete set of inference rules and recursive transformation functions* considering all types of GAV inter-schema correspondences supported by the model. On the other hand, filter expression rewriting is based on ontology term substitution using 1:1 cardinality mappings. Mapping relationships like equivalence and subsumption do not affect the query rewriting process but the evaluation results obtained by executing the reformulated query over the integrated RDF data sources.

After the rewriting of the input query graph pattern using the predefined ontology mappings, *the algorithm performs query decomposition* by grouping the resulted triple patterns and filter expressions and enclosing them in SERVICE clauses based on the appearing ontology terms and the integrated data source endpoints. The resulted federated queries can be executed directly on any federated query engine, or exploited as logical query plans by any ontology based mediator system.

The presented query rewriting algorithms and transformation functions have been evaluated for their soundness and completeness, and are *proved to provide semantics preserving queries* with respect to the GAV mapping types supported by the mapping model. Existing LAV/GLAV query rewriting algorithms, like *Bucket* [52], *Minicon* [71, 70] and *Inverse-Rules* [24], can be relatively easily adapted using the proposed inference rules and recursive transformation functions.

Chapter 6

Evaluation and Use

The SPARQL–RW Framework has been fully implemented using J2SE as a software platform, OWL API [39] for supporting the mapping inference and inconsistency identification tasks, and Jena ARQ[1] for SPARQL query parsing and manipulation. In this chapter we describe the experimental evaluation conducted on SPARQL–RW in terms of its query rewriting efficiency and we discuss the obtained results. Moreover, we present the Semantic Query Mediation Prototype Infrastructure that we have developed in order to demonstrate the applicability of the SPARQL–RW Framework in a real data integration scenario, involving DBpedia and several biodiversity data providers.

6.1 Experimental Evaluation

Query rewriting efficiency is a strong requirement for any system performing query mediation. Compared to mapping inference and inconsistency identification which are mainly carried out on system setup and/or in a regular scheduled basis, query rewriting needs to be performed on every query submission in the mediator. In this section we describe the experimental evaluation conducted on SPARQL–RW and we discuss the efficiency of its query rewriting algorithms.

Considering that SPARQL queries are mainly consisted of triple patterns and much less filter expressions, the query rewriting time depends on two factors: (*a*) the *number of triple patterns in the input query*, and (*b*) the *number of operations and ontology terms (nodes) in the exploited mappings*. The number of the integrated data sources is directly reflected in the size and complexity of the exploited GAV mappings, and thus, it is not considered as an additional factor. To evaluate the efficiency of the rewriting process, we measured the time required by SPARQL–RW to rewrite input queries of different size and type, using mappings of varying complexity in a hypothetical data integration scenario. The evaluation was performed on an Intel Core i7 processor at 2.67GHz, with 4GB RAM, running Linux and Java 1.7.

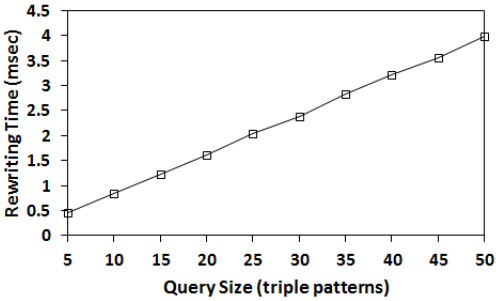
[1] Jena ARQ: `https://jena.apache.org/documentation/query/index.html`

In our experiment, we created 20 different SPARQL queries by modifying the number and type of the triple patterns appearing in their graph pattern. 10 out of the 20 queries were consisted of data triple patterns, while the rest were consisted of schema triple patterns. Additionally, we generated several mappings of different complexity by varying the number of the appearing operations and ontology terms (nodes). Aiming to measure the rewriting time of the worst case scenario, the generated mappings did not contain any unsupported schema operations, ensuring that mapping simplification in schema triple pattern rewriting would not lead to shorter mappings.

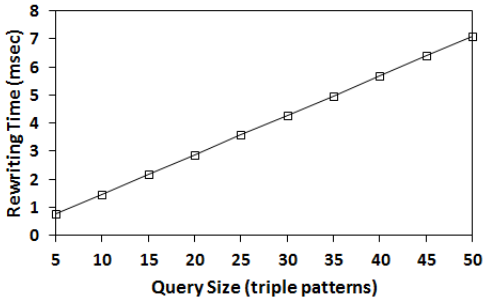
Table 6.1 presents the time required for the rewriting of every generated query using mappings of different size ($1, 3, 5, 7, 10, 15, 25, 40$ nodes). Similarly, Figure 6.1 presents the rewriting times of data triple pattern queries in the form of a diagram. Considering the obtained results, the SPARQL–RW rewriting method is proved to be fast, even for highly complex queries and ontology mappings. More specifically, the algorithm requires less than 70 msec to rewrite queries consisted of 50 triple patterns, using mappings of size 40 nodes. Additionally, note that the rewriting time increases linearly for mappings of the same complexity and input queries of increasing size.

Table 6.1: Query rewriting time (in milliseconds), varying the input query size and mapping complexity.

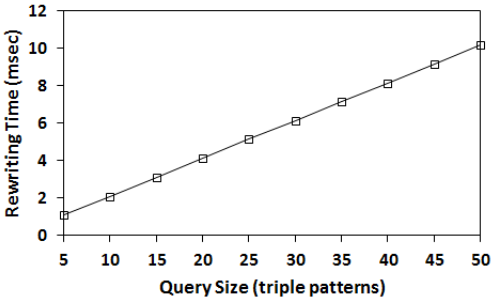
	Number of	**Number of Nodes in the Exploited Mappings**							
	Triple Patterns	**1**	**3**	**5**	**7**	**10**	**15**	**25**	**40**
Data Triple Pattern Queries	**5**	0.467	0.766	1.08	1.414	1.845	2.571	4.171	6.6
	10	0.847	1.445	2.084	2.728	3.619	5.118	8.19	13.035
	15	1.233	2.195	3.102	3.993	5.38	7.676	12.348	19.655
	20	1.617	2.88	4.113	5.305	7.133	10.26	16.428	26.21
	25	2.046	3.569	5.133	6.636	8.905	12.771	20.489	32.747
	30	2.389	4.27	6.116	7.984	10.773	15.407	24.653	39.162
	35	2.826	4.951	7.121	9.244	12.568	17.946	28.756	45.881
	40	3.21	5.665	8.123	10.558	14.21	20.381	32.695	52.396
	45	3.556	6.415	9.16	11.919	15.879	23.125	36.768	58.671
	50	3.978	7.09	10.164	13.285	17.573	25.576	41.08	64.73
Schema Triple Pattern Queries	**5**	0.493	0.8	1.083	1.393	1.84	2.612	4.145	6.562
	10	0.863	1.471	2.093	2.705	3.636	5.162	8.167	13.107
	15	1.26	2.195	3.111	4.029	5.388	7.654	12.157	19.701
	20	1.642	2.898	4.095	5.333	7.182	10.226	16.143	26.192
	25	2.033	3.58	5.129	6.681	8.932	12.818	20.384	32.794
	30	2.424	4.308	6.147	7.997	10.746	15.36	24.434	39.329
	35	2.816	5.023	7.204	9.237	12.514	17.898	28.379	45.9
	40	3.217	5.677	8.163	10.533	14.266	20.445	32.574	52.383
	45	3.61	6.399	9.218	11.914	16.065	23.014	36.653	58.74
	50	3.994	7.107	10.111	13.29	17.858	25.619	40.722	65.26



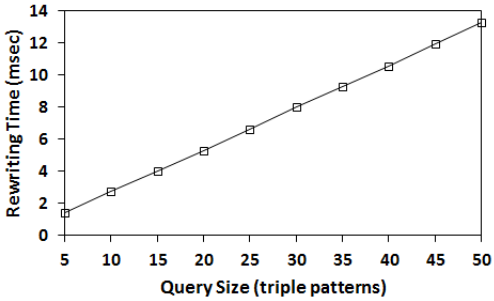
(a) $n = 1$



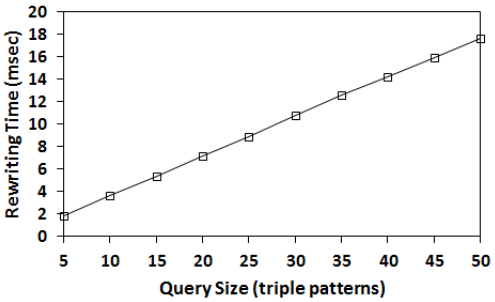
(b) $n = 3$



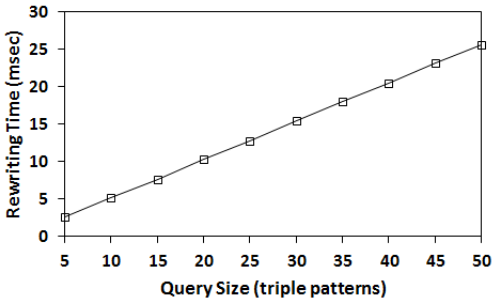
(c) $n = 5$



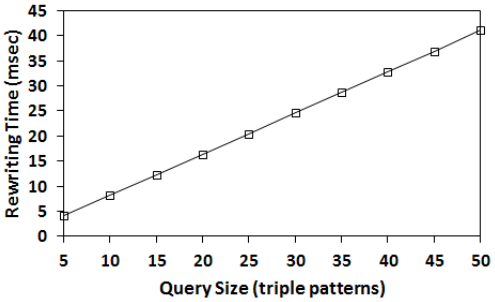
(d) $n = 7$



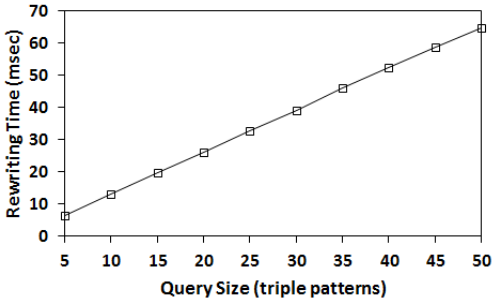
(e) $n = 10$



(f) $n = 15$



(g) $n = 25$



(h) $n = 40$

Figure 6.1: Rewriting time vs. query size for queries consisting of data triple patterns and mappings containing $n = 1, 3, 5, 7, 10, 15, 25$ and 40 nodes (operations and ontology terms).

Figure 6.2 shows the data and schema triple pattern rewriting times, for queries of different sizes and mappings consisted of 50 nodes (both operations and ontology terms). As expected, the use of different rules and transformation functions in the rewriting of data and schema queries had no noticeable effect in the query rewriting time. Finally, Figure 6.3 presents the impact of increasing the complexity of the exploited mappings for queries of different sizes consisting of data triple patterns. As shown in the diagram, the increase in mapping complexity is reflected in the query rewriting time, not only for large but for smaller queries also. Considering, however, that an average SPARQL query is consisted of $10 - 15$ triple patterns, an increase in mapping complexity by 5 operations and ontology terms affects the rewriting time only by $1.5 - 2.4$ msec (see Table 6.1).

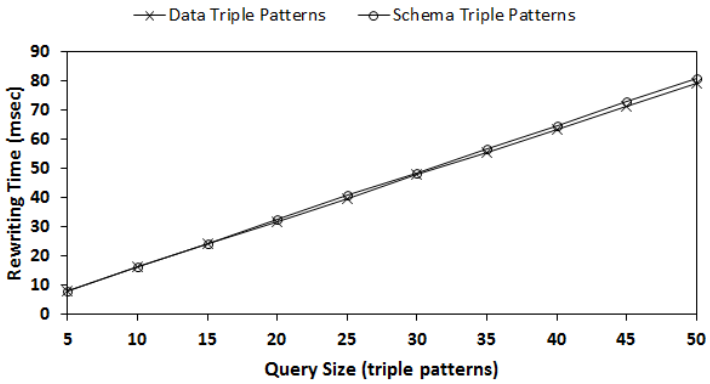


Figure 6.2: Data vs. schema triple pattern rewriting times, for queries of different sizes and mappings consisting of 50 nodes (operations and ontology terms).

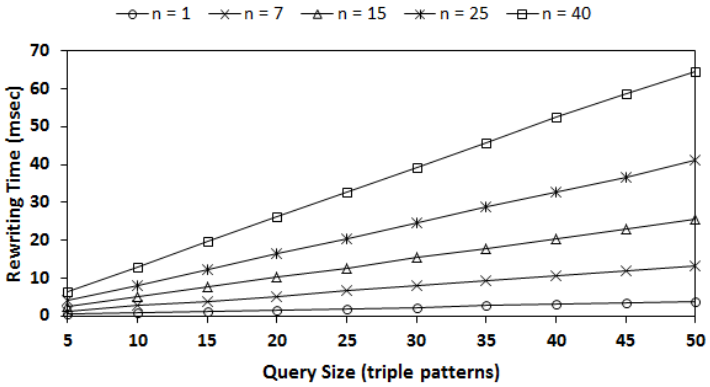


Figure 6.3: The impact of increasing the mapping complexity ($n = 1, 7, 15, 25$ and 40 operations/ontology terms), for queries of different sizes consisting of data triple patterns.

6.2 The Semantic Query Mediation Prototype Infrastructure

This section presents the Semantic Query Mediation Prototype Infrastructure that we have developed in order to demonstrate the applicability of the SPARQL–RW Framework. The Semantic Query Mediation Prototype Infrastructure (SQMPI) is a mediator system offering transparent query access over federated RDF data sources. It is able to answer SPARQL queries expressed in terms of a global ontology schema, using information from a set of integrated data sources. The SQMPI functionality is exposed through a web application, implemented using the Spring MVC Framework. Figure 6.4 presents the system architecture focusing on the server-side modules and on their communication with the integrated data sources.

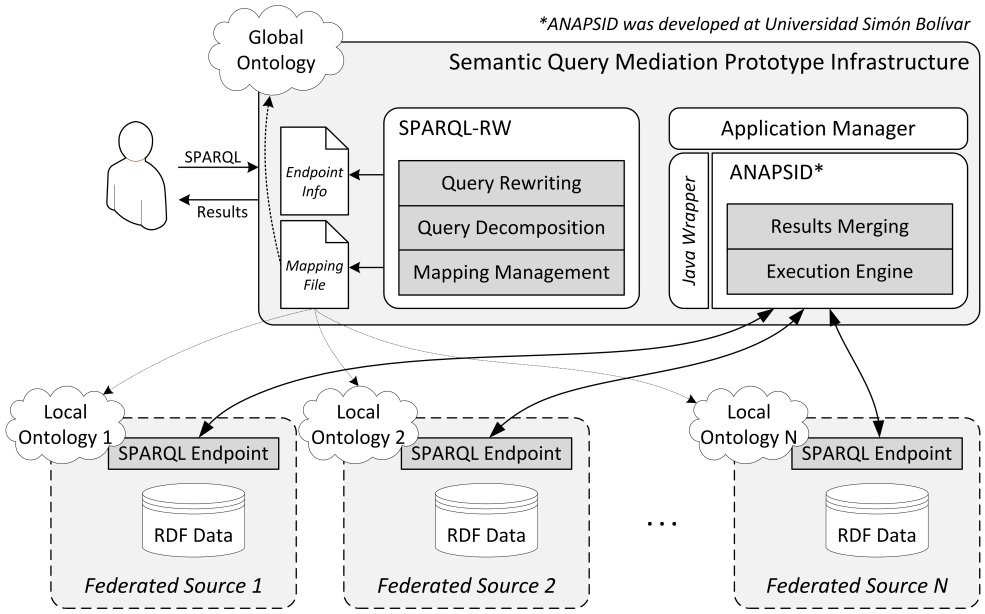


Figure 6.4: The SQMPI architecture.

SQMPI employs SPARQL–RW both for mapping management, as well as for the rewriting and decomposition of SPARQL queries submitted to the mediator. Query rewriting is based on a set of predefined GAV ontology mappings between the global (mediator) schema and the schemas of the integrated data sources, while query decomposition relies on the available data source endpoints. The module is initialized during the system's start-up using configuration files indicating the available mappings and SPARQL endpoints.

For query execution and results merging, SQMPI employs *ANAPSID*. ANAPSID [2] is an open source query engine for SPARQL endpoints, developed at Universidad Simón Bolívar. ANAPSID adapts query execution schedulers to data availability and run-time conditions. It provides physical SPARQL operators that detect when a source becomes blocked or data traffic is bursty, and opportunistically, the operators produce results as quickly as data arrives from the sources.

Due to the fact that ANAPSID has been implemented in Python, we developed a Java Wrapper on top of it, encapsulating the ANAPSID's internal functionality. The *ANAPSID Java Wrapper* exposes services for query submission and result set streaming. Both SPARQL–RW and ANAPSID Java Wrapper are administered by the *Application Manager* module. Application Manager acts as a centralized point of control, being responsible for the analysis of the received requests in the server side and handling the communication between the aforementioned modules.

At run time, when a SPARQL query is submitted to the mediator, it is processed, reformulated and decomposed by SPARQL–RW. The resulted query is subsequently passed to the ANAPSID Java Wrapper through the Application Manager. The ANAPSID Java Wrapper transfers the reformulated federated query to ANAPSID in order to be executed over the integrated data sources. The returned results are merged by ANAPSID and returned to the Application Manager through the ANAPSID Java Wrapper. Finally, the Application Manager forwards the results to the client side for presentation purposes.

6.2.1 Case Study: Integrating Natural Europe and DBpedia

The Semantic Query Mediation Prototype Infrastructure has been successfully deployed and tested in a real data integration scenario involving DBpedia, as well as 6 biodiversity data providers from the Natural Europe project. The prototype mediator system is currently available at `http://kostasmakris.com/sparql-rw`, while Figures 6.5, 6.6 and 6.7 present some indicative screenshots of the system's graphical user interface.

Natural Europe [1] is a project co-funded by EC under the ICT-PSP programme offering, among others, appropriate tools and services that allow natural history museums (NHMs) to: (*a*) uniformly describe and semantically annotate their content according to international standards and specifications, (*b*) interconnect their digital libraries, and (*c*) expose their scientific collections to cultural and biodiversity networks, as well as to the Linked Data community [83]. The Natural Europe infrastructure has been currently deployed in 6 European museums, allowing curators to publish, semantically describe, manage and disseminate a large volume of cultural heritage objects (CHOs), including animal, plant and mineral specimens. All the contributed CHO descriptions have been published as Linked Data, in separate Virtuoso servers, conforming to the Natural Europe Ontology[2] and can be accessed through 6 SPARQL endpoints. Table 6.2 presents the current number of CHOs and RDF triples, published in each federated museum node.

DBpedia [9], on the other hand, is a knowledge base currently describing about 4.0 million things including 832,000 persons, 639,000 places, and 226,000 species. The dataset has been created by extracting structured data from Wikipedia and has been classified in a consistent cross-domain ontology[3]. Apart from web services, data can be accessed through a SPARQL endpoint.

[2]Natural Europe Ontology: `http://natural-europe.tuc.gr/ne-ontology-v01.owl`
[3]DBpedia Ontology: `http://wiki.dbpedia.org/Ontology39?v=g9b`

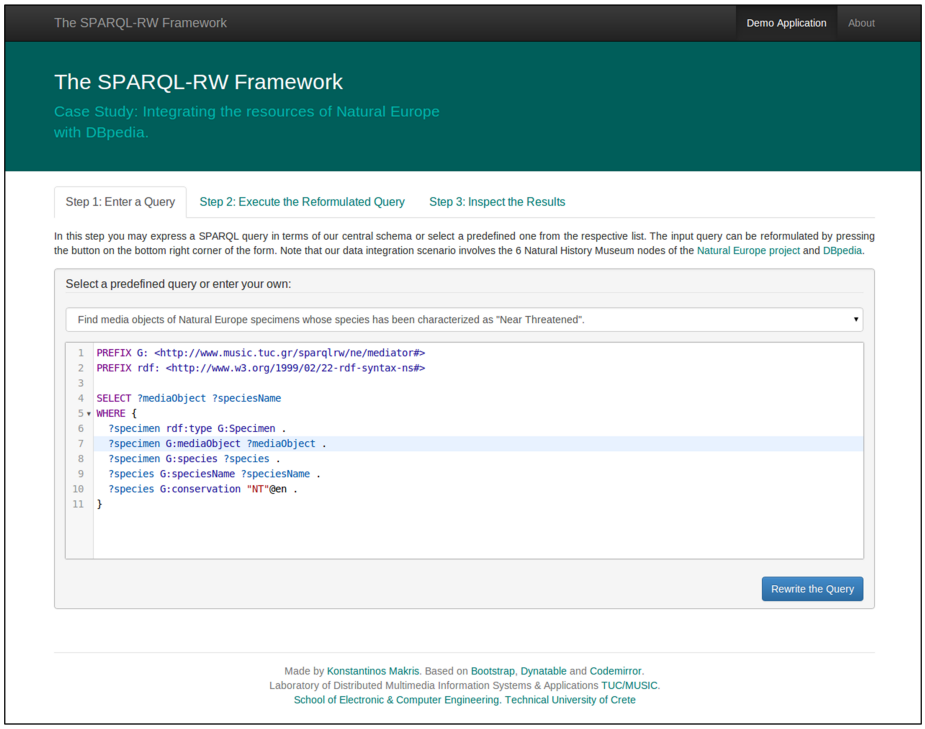


Figure 6.5: Screenshot presenting the graphical user interface of the SQMPI web application (1/3).

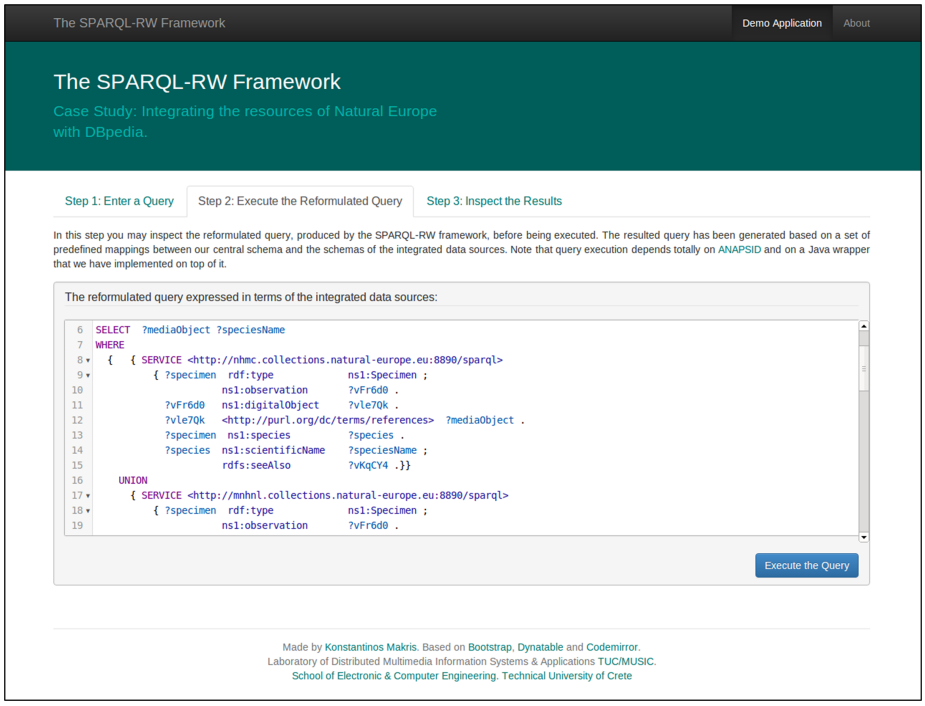


Figure 6.6: Screenshot presenting the graphical user interface of the SQMPI web application (2/3).

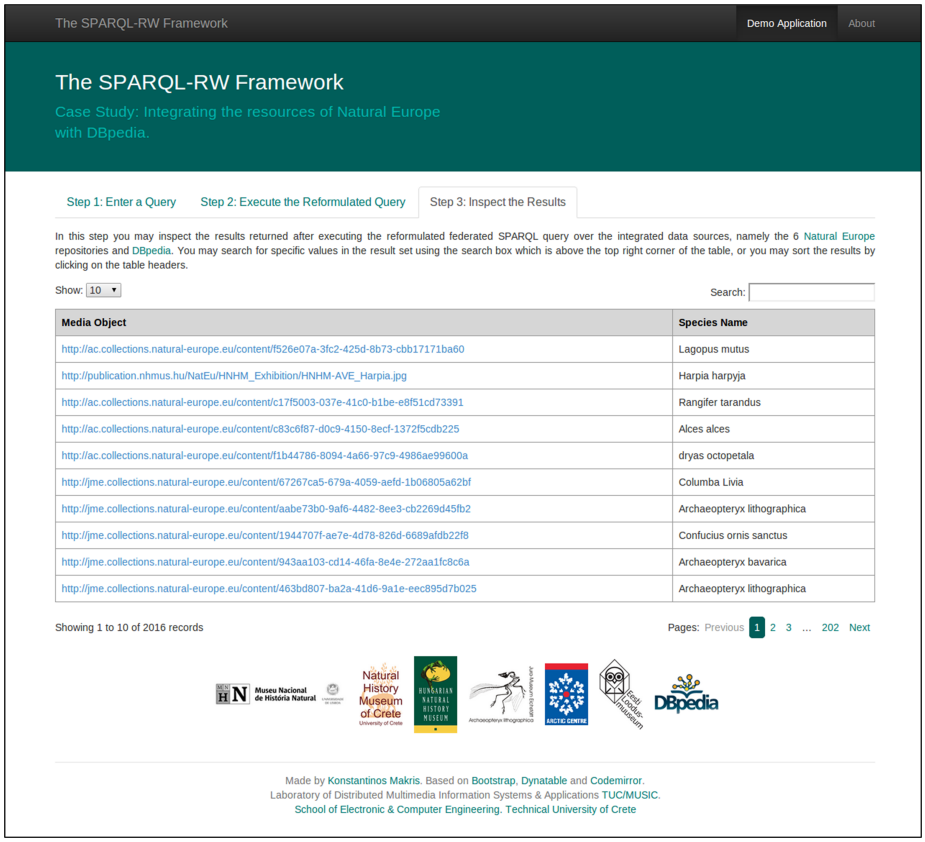


Figure 6.7: Screenshot presenting the graphical user interface of the SQMPI web application (3/3).

Table 6.2: The current number of cultural heritage objects (CHOs) and RDF triples, published in each Natural Europe federated museum node.

Natural Europe Federated Data Sources	**CHO Number**	**RDF Triples**
Natural History Museum of Crete (NHMC)	4,010	195,905
National Museum of Natural History of Lisbon (MNHNL)	2,686	115,913
Jura-Museum Eichstätt (JME)	1,658	60,371
Arctic Center (AC)	480	18,715
Hungarian Natural History Museum (HNHM)	4,244	154,543
Estonian Museum of Natural History (TNHM)	1,972	85,773
Total	15,050	631,220

The integration of the Natural Europe federated data sources and DBpedia, under a common conceptualization, allows uniform information access and enables the execution of highly sophisticated queries combining the persisted knowledge. As an example, specimen and media object information persisted in the Natural Europe repositories can be combined with DBpedia species conservation status information, in order to answer queries like the following: *"Find photos of Natural Europe specimens whose species have been characterized as near threatened"*. Furthermore, external applications and end-users are able to interact with a single ontology schema and endpoint, without having to be aware of the schemas and SPARQL endpoints of the integrated data sources. As a result, highly complex queries, combining information from multiple data sources, can be expressed in a few triples, reducing drastically the effort of query composition.

6.3 Summary

In this chapter we described the experimental evaluation conducted on SPARQL–RW in terms of its query rewriting efficiency and we discussed the obtained results. To evaluate the efficiency of the rewriting process, we measured the time required by SPARQL–RW to rewrite input queries of different size and type, using mappings of varying complexity. The experimental evaluation proved that the SPARQL–RW query rewriting method is extremely fast, even for highly complex queries and ontology mappings. The rewriting time increases linearly for mappings of the same complexity and input queries of increasing size. Furthermore, as expected, the use of different rules and transformation functions in the rewriting of data and schema queries has no noticeable effect in the query rewriting time.

Additionally to the experimental evaluation, this chapter presented the Semantic Query Mediation Prototype Infrastructure that we have developed in order to demonstrate the applicability of the SPARQL–RW Framework in a real data integration scenario. Our prototype mediator system integrates DBpedia, as well as 6 biodiversity data providers from the Natural Europe project. It is based on SPARQL–RW for mapping management, query rewriting and query decomposition, and enables the execution of highly sophisticated queries. External applications and end-users are able to interact with a single ontology schema and endpoint, and thus, highly complex queries, combining information from multiple data sources, can be expressed in a few triples.

Chapter 7

Conclusions and Future Research

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. In this thesis we presented SPARQL–RW, a Framework providing transparent query access over mapped RDF data sources by supporting mapping modeling and query rewriting in the context of ontology based mediators.

In more detail, we proposed a model for the expression of mappings between ontology schemas. The mapping model consists of a grammar defining the mapping types which can be exploited in SPARQL query rewriting, as well as a specification of the mapping type semantics. It is based on Description Logics and it is capable of describing a great variety of OWL inter-schema correspondences providing *high flexibility* and *satisfying different system requirements and user needs*. Furthermore, it is able to support well-known mapping formalisms, including *GAV*, *LAV*, and *GLAV*, satisfying strong data integration requirements for *query rewriting efficiency* and *extensibility to new sources*. To the best of our knowledge, there is no system supporting these formalisms in the context of ontology based mediator architectures.

Additionally to the mapping model, we defined a language based on XML syntax, being able to represent the discussed mapping types and formalisms. The mapping language combines a set of criteria including *simplicity*, *expressiveness*, *executability*, and *schema language agnosticism*. Furthermore, the use of XML Schema in mapping language definition: (*a*) provides exceptional *validation* capabilities, (*b*) supports easy mapping *serialization and deserialization*, and (*c*) enables *interoperability* with external systems and applications.

Aiming to assist the mapping definition process and support the maintenance of mappings conforming to the SPARQL–RW mapping model, we provided methods for performing mapping inference and identifying inconsistencies in a given set of mappings and ontology schemas. Both methods exploit the underlying DL semantics of the SPARQL–RW mapping model and are based on the use of well-known reasoning techniques.

Regarding query rewriting, we provided a formal method for the reformulation of SPARQL queries posed over the mediator, into federated queries referring to the integrated data sources. The method *considers the SPARQL 1.1 specification and it is able to handle both data and schema queries*. The proposed query rewriting strategy is *based on a complete set of inference rules and recursive transformation functions*, and relies on the reformulation of the input query graph pattern using a set of predefined GAV ontology mappings and data source endpoints. The resulted queries can be executed directly on any federated query engine, or exploited as logical query plans by any ontology based mediator system. The provided algorithms and transformation functions have been formally evaluated for their soundness and completeness, and are *proved to provide semantics preserving queries* with respect to the GAV inter-schema correspondences supported by the model. To the extent of our knowledge, there is no system performing SPARQL 1.1 query rewriting in general, or SPARQL query rewriting by exploiting well-known mapping formalisms in the field of ontology based mediators.

The SPARQL–RW Framework has been fully implemented and evaluated in terms of its query rewriting efficiency, measuring the time required for the reformulation of queries of different size and type, using mappings of varying complexity. The experimental evaluation proved that the SPARQL–RW query rewriting process is extremely fast, even for highly complex queries and ontology mappings. Furthermore, the Framework has been tested in a prototype mediator system that we have developed, supporting the integration of DBpedia [9] and several biodiversity data providers from the Natural Europe project [1]. This infrastructure enables the execution of highly sophisticated queries combining specimen and media object information persisted in the Natural Europe repositories with species information available in DBpedia. External applications and end-users are able to interact with a single ontology schema and endpoint, without having to be aware of the schemas and SPARQL endpoints of the integrated data sources. As a result, highly complex queries, combining information from multiple data sources, can be expressed in a few triples, reducing drastically the effort of query composition.

Our current research focuses on the development of LAV and GLAV query rewriting methods, combining the SPARQL–RW query rewriting rules and transformation functions with well-known algorithms such as *Bucket* [52], *Minicon* [71, 70] and *Inverse-Rules* [24]. Furthermore, we aim to develop an interactive graphical tool supporting the specification of mappings conforming to the SPARQL–RW mapping model. The tool will be using the SPARQL–RW mapping inference and inconsistency identification techniques for automating mapping generation and supporting mapping maintenance. Mapping representation and serialization will be based on the SPARQL–RW mapping language. Finally, we plan to integrate SPARQL–RW with the XS2OWL [88] and SPARQL2XQuery [8] Frameworks, in order to enable access to heterogeneous web repositories.

Bibliography

[1] Natural Europe. `http://www.natural-europe.eu/`.

[2] M. Acosta, M.-E. Vidal, T. Lampo, J. Castillo, and E. Ruckhaus. ANAPSID: An Adaptive Query Processing Engine for SPARQL Endpoints. In *Proceedings of the 10th International Conference on The Semantic Web - Volume Part I*, ISWC'11, pages 18–34, Berlin, Heidelberg, 2011. Springer-Verlag.

[3] B. Alexe, B. ten Cate, P. G. Kolaitis, and W.-C. Tan. Designing and Refining Schema Mappings via Data Examples. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 133–144, New York, NY, USA, 2011. ACM.

[4] M. Arenas and J. Pérez. Querying Semantic Web Data with SPARQL. In *Proceedings of the 30th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '11, pages 305–316, New York, NY, USA, 2011. ACM.

[5] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.

[6] C. Beeri, A. Y. Levy, and M.-C. Rousset. Rewriting Queries Using Views in Description Logics. In *Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '97, pages 99–108, New York, NY, USA, 1997. ACM.

[7] Z. Bellahsene, A. Bonifati, and E. Rahm, editors. *Schema Matching and Mapping*. Springer, 2011.

[8] N. Bikakis, C. Tsinaraki, I. Stavrakantonakis, N. Gioldasis, and S. Christodoulakis. The SPARQL2XQuery Interoperability Framework. *CoRR*, 2013.

[9] C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, and S. Hellmann. DBpedia - A Crystallization Point for the Web of Data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(3):154–165, 2009.

[10] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor. Freebase: A Collaboratively Created Graph Database for Structuring Human Knowledge. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1247–1250, New York, NY, USA, 2008. ACM.

[11] P. Bouquet, F. Giunchiglia, F. Harmelen, L. Serafini, and H. Stuckenschmidt. C-OWL: Contextualizing Ontologies. In *Proceedings of the 2nd International Semantic Web Conference*, volume 2870 of *ISWC '03*, pages 164–179. Springer, Berlin–Heidelberg, Germany, 2003.

[12] P. Bouquet, L. Serafini, S. Zanobini, and S. Sceffer. Bootstrapping Semantics on the Web: Meaning Elicitation from Schemas. In *Proceedings of the 15th International Conference on World Wide Web*, WWW '06, pages 505–512, New York, NY, USA, 2006. ACM.

[13] D. Brickley and R. V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. *W3C Recommendation*, 10, 2004.

[14] C. Buil-Aranda, M. Arenas, O. Corcho, and A. Polleres. Federating Queries in SPARQL 1.1: Syntax, Semantics and Evaluation. *Web Semant.*, 18(1):1–17, Jan. 2013.

[15] D. Calvanese, G. De Giacomo, M. Lenzerini, and R. Rosati. View-based Query Answering in Description Logics: Semantics and Complexity. *J. Comput. Syst. Sci.*, 78(1):26–46, Jan. 2012.

[16] D. Calvanese, G. D. Giacomo, and M. Lenzerini. Answering Queries Using Views over Description Logics Knowledge Bases. In *Proceedings of the 17th National Conference on Artificial Intelligence*, AAAI '00, pages 386–391. AAAI Press, 2000.

[17] B. T. Cate, V. Dalmau, and P. G. Kolaitis. Learning Schema Mappings. *ACM Trans. Database Syst.*, 38(4):28:1–28:31, Dec. 2013.

[18] N. Choi, I.-Y. Song, and H. Han. A Survey on Ontology Mapping. *SIGMOD Rec.*, 35(3):34–41, Sept. 2006.

[19] G. Correndo, M. Salvadores, I. Millard, H. Glaser, and N. Shadbolt. SPARQL Query Rewriting for Implementing Data Integration over Linked Data. In *Proceedings of the 1st International Workshop on Data Semantics*, DataSem '10, pages 4:1–4:11, New York, NY, USA, 2010. ACM.

[20] J. David, J. Euzenat, F. Scharffe, and C. Trojahn dos Santos. The Alignment API 4.0. *Semant. web*, 2(1):3–10, Jan. 2011.

[21] A. Doan, A. Halevy, and Z. Ives. *Principles of Data Integration*. Morgan Kaufmann Publishers Inc., 1st edition, July 2012.

[22] M. Doerr, C.-E. Ore, and S. Stead. The CIDOC Conceptual Reference Model: A New Standard for Knowledge Sharing. In *Tutorials, Posters, Panels and Industrial Contributions at the 26th International Conference on Conceptual Modeling*, volume 83 of *ER '07*, pages 51–56, Darlinghurst, Australia, Australia, 2007. Australian Computer Society, Inc.

[23] M. J. Dürst and M. Suignard. Internationalized Resource Identifiers (IRIs). Internet RFC 3987, January 2005.

[24] O. M. Duschka and M. R. Genesereth. Answering Recursive Queries Using Views. In *Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '97, pages 109–116, New York, NY, USA, 1997. ACM.

[25] J. Euzenat. An API for Ontology Alignment. In *Proceedings of the 3rd International Semantic Web Conference*, volume 3298 of *ISWC '04*, pages 698–712, Berlin, Heidelberg, Nov. 2004. Springer.

[26] J. Euzenat and P. Shvaiko. *Ontology Matching*. Springer-Verlag, Berlin-Heidelberg, 2 edition, 2013.

[27] R. Fagin, P. G. Kolaitis, L. Popa, and W. C. Tan. Schema Mapping Evolution Through Composition and Inversion. In Z. Bellahsene, A. Bonifati, and E. Rahm, editors, *Schema Matching and Mapping*, pages 191–222. Springer, 2011.

[28] T. Fujino and N. Fukuta. A SPARQL Query Rewriting Approach on Heterogeneous Ontologies with Mapping Reliability. In *Proceedings of the 2012 IIAI International Conference on Advanced Applied Informatics*, IIAI-AAI '12, pages 230–235, Washington, DC, USA, 2012. IEEE Computer Society.

[29] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. The TSIMMIS project: Integation of Heterogeneous Information Sources. *Journal of Intelligent Information Systems*, 8(2):117–132, 1997.

[30] P. Gearon, A. Passant, and A. Polleres. SPARQL 1.1 Update. *W3C Recommendation*, March 2013.

[31] O. Görlitz and S. Staab. Federated Data Management and Query Optimization for Linked Open Data. In *New Directions in Web Data Management 1*, volume 331 of *Studies in Computational Intelligence*, pages 109–137. 2011.

[32] A. Halevy. Answering Queries using Views – A Survey. *VLDB Journal*, 10(4):270–294, 2001.

[33] A. Y. Halevy. Theory of Answering Queries Using Views. *SIGMOD Rec.*, 29:40–47, December 2000.

[34] S. Harris and A. Seaborne. SPARQL 1.1 Query Language. *W3C Recommendation*, March 2013.

[35] O. Hartig. An Overview on Execution Strategies for Linked Data Queries. *Datenbank-Spektrum*, 13(2):89–99, 2013.

[36] O. Hartig, C. Bizer, and J.-C. Freytag. Executing SPARQL Queries over the Web of Linked Data. In *Proceedings of the 8th International Semantic Web Conference*, ISWC '09, pages 293–309, Berlin, Heidelberg, 2009. Springer-Verlag.

[37] P. Hayes. RDF Semantics. *W3C Recommendation*, February 2004.

[38] P. Hitzler, M. Krötzsch, and S. Rudolph. *Foundations of Semantic Web Technologies*. CRC Press, 2010.

[39] M. Horridge and S. Bechhofer. The OWL API: A Java API for OWL Ontologies. *Semant. web*, 2(1):11–21, Jan. 2011.

[40] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. *W3C Member Submission*, May 2004.

[41] Z. G. Ives, A. Y. Halevy, P. Mork, and I. Tatarinov. Piazza: mediation and integration infrastructure for Semantic Web data. *J. Web Sem.*, 1(2):155–175, 2004.

[42] E. Jiménez-Ruiz, B. Cuenca Grau, I. Horrocks, and R. Berlanga. Ontology Integration Using Mappings: Towards Getting the Right Logical Consequences. In *Proceedings of the 6th European Semantic Web Conference on The Semantic Web: Research and Applications*, ESWC '09, pages 173–187, Berlin, Heidelberg, 2009. Springer-Verlag.

[43] Y. Kalfoglou and M. Schorlemmer. Ontology Mapping: The State of the Art. *Knowledge Engineering Review*, 18(1):1–31, Jan. 2003.

[44] A. Kalyanpur, B. Parsia, E. Sirin, and J. Hendler. Debugging Unsatisfiable Classes in OWL Ontologies. *Web Semant.*, 3(4):268–293, Dec. 2005.

[45] G. Konstantinidis and J. L. Ambite. Scalable Query Rewriting: A Graph-based Approach. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 97–108, New York, NY, USA, 2011. ACM.

[46] G. Konstantinidis and J. L. Ambite. Optimizing Query Rewriting for Multiple Queries. In *Proceedings of the 9th International Workshop on Information Integration on the Web*, IIWeb '12, pages 7:1–7:6, New York, NY, USA, 2012. ACM.

[47] N. Koul and V. Honavar. Learning in Presence of Ontology Mapping Errors. In *Proceedings of the 2010 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*, WI-IAT '10, pages 291–296, Washington, DC, USA, 2010. IEEE Computer Society.

[48] J. Kunze and T. Baker. The Dublin Core Metadata Element Set. RFC 5013, August 2007.

[49] W. Le, S. Duan, A. Kementsietsidis, F. Li, and M. Wang. Rewriting Queries on SPARQL Views. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, pages 655–664, New York, NY, USA, 2011. ACM.

[50] M. Lenzerini. Data Integration: A Theoretical Perspective. In *Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, pages 233–246, New York, NY, USA, 2002. ACM.

[51] A. Y. Levy. *Logic-Based Techniques in Data Integration*, pages 575–595. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[52] A. Y. Levy, A. Rajaraman, and J. J. Ordille. Querying Heterogeneous Information Sources Using Source Descriptions. In *Proceedings of the 22th International Conference on Very Large Data Bases*, VLDB '96, pages 251–262, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[53] F. L. R. Lopes, E. R. Sacramento, and B. F. Loscio. Using Heterogeneous Mappings for Rewriting SPARQL Queries. In *Proceedings of the 23rd International Workshop on Database and Expert Systems Applications*, DEXA '12, pages 267–271, Washington, DC, USA, 2012. IEEE Computer Society.

[54] J. Madhavan and A. Y. Halevy. Composing Mappings Among Data Sources. In *Proceedings of the 29th International Conference on Very Large Data Bases*, VLDB '03, pages 572–583. VLDB Endowment, 2003.

[55] A. Maedche, B. Motik, N. Silva, and R. Volz. MAFRA - A MApping FRAmework for Distributed Ontologies. In *Proceedings of the 13th International Conference on Knowledge Engineering and Knowledge Management. Ontologies and the Semantic Web*, EKAW '02, pages 235–250, London, UK, UK, 2002. Springer-Verlag.

[56] F. Manola and E. Miller. RDF Primer. *W3C Recommendation*, 10:1–107, 2004.

[57] D. L. McGuinness and F. Van Harmelen. OWL Web Ontology Language Overview. *W3C Recommendation*, 10:1–19, 2004.

[58] C. Meilicke. The Relevance of Reasoning and Alignment Incoherence in Ontology Matching. In *Proceedings of the 6th European Semantic Web Conference on The Semantic Web: Research and Applications*, ESWC '09, pages 934–938, Berlin, Heidelberg, 2009. Springer-Verlag.

[59] C. Meilicke, H. Stuckenschmidt, and A. Tamilin. Improving Automatically Created Mappings Using Logical Reasoning. In *Proceedings of the 2006 International Workshop on Ontology Matching*, volume 225 of *OM '06*. CEUR-WS.org, 2006.

[60] C. Meilicke, H. Stuckenschmidt, and A. Tamilin. Repairing Ontology Mappings. In *Proceedings of the 22nd National Conference on Artificial Intelligence*, AAAI'07, pages 1408–1413. AAAI Press, 2007.

[61] C. Meilicke, H. Stuckenschmidt, and A. Tamilin. Reasoning Support for Mapping Revision. *J. Log. and Comput.*, 19(5):807–829, Oct. 2009.

[62] G. Montoya, M.-E. Vidal, O. Corcho, E. Ruckhaus, and C. Buil-Aranda. Benchmarking Federated SPARQL Query Engines: Are Existing Testbeds Enough? In *Proceedings of the 11th International Conference on The Semantic Web - Volume Part II*, ISWC'12, pages 313–324, Berlin, Heidelberg, 2012. Springer-Verlag.

[63] B. Motik, B. C. Grau, I. Horrocks, Z. Wu, A. Fokoue, and C. Lutz. OWL 2 Web Ontology Language: Profiles. *W3C Recommendation*, 2009.

[64] B. Motik, P. F. Patel-Schneider, B. Parsia, C. Bock, A. Fokoue, P. Haase, R. Hoekstra, I. Horrocks, A. Ruttenberg, U. Sattler, and M. Smith. OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax. *W3C Recommendation*, 2009.

[65] N. F. Noy. Semantic Integration: A Survey of Ontology-based Approaches. *SIGMOD Rec.*, 33(4):65–70, Dec. 2004.

[66] P. F. Patel-Schneider, P. Hayes, and I. Horrocks. OWL Web Ontology Language Semantics and Abstract Syntax. *W3C Recommendation*, February 2004.

[67] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and Complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3):16:1–16:45, Sept. 2009.

[68] A. Polleres. From SPARQL to Rules (and Back). In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 787–796, New York, NY, USA, 2007. ACM.

[69] A. Polleres, F. Scharffe, and R. Schindlauer. SPARQL++ for Mapping Between RDF Vocabularies. In *Proceedings of the 2007 OTM Confederated International Conference on On the Move to Meaningful Internet Systems: CoopIS, DOA, ODBASE, GADA, and IS - Volume Part I*, OTM'07, pages 878–896, Berlin, Heidelberg, 2007. Springer-Verlag.

[70] R. Pottinger and A. Halevy. MiniCon: A Scalable Algorithm for Answering Queries Using Views. *The VLDB Journal*, 10(2-3):182–198, Sept. 2001.

[71] R. Pottinger and A. Y. Levy. A Scalable Algorithm for Answering Queries Using Views. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 484–495, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[72] E. Prud'hommeaux and C. Buil-Aranda. SPARQL 1.1 Federation Extensions. *W3C Recommendation*, March 2013.

[73] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. *W3C Recommendation*, 4:1–106, 2008.

[74] B. Quilitz and U. Leser. Querying Distributed RDF Data Sources with SPARQL. In *Proceedings of the 5th European Semantic Web Conference on The Semantic Web: Research and Applications*, ESWC'08, pages 524–538, Berlin, Heidelberg, 2008. Springer-Verlag.

[75] C. R. Rivero, I. Hernández, D. Ruiz, and R. Corchuelo. Mosto: Generating SPARQL Executable Mappings Between Ontologies. In *Proceedings of the 30th International Conference on Advances in Conceptual Modeling: Recent Developments and New Directions*, ER'11, pages 345–348, Berlin, Heidelberg, 2011. Springer-Verlag.

[76] M. Saleem, Y. Khan, A. Hasnain, I. Ermilov, and A.-C. Ngonga Ngomo. A Fine-Grained Evaluation of SPARQL Endpoint Federation Systems. *Under-review: Semantic Web Journal*, 2014.

[77] B. Sanghvi, N. Koul, and V. Honavar. Identifying and Eliminating Inconsistencies in Mappings Across Hierarchical Ontologies. In *Proceedings of the 2010 International Conference on On the Move to Meaningful Internet Systems: Part II*, OTM'10, pages 999–1008, Berlin, Heidelberg, 2010. Springer-Verlag.

[78] M. Schmidt, O. Görlitz, P. Haase, G. Ladwig, A. Schwarte, and T. Tran. FedBench: A Benchmark Suite for Federated Semantic Data Query Processing. In *Proceedings of the 10th International Conference on The Semantic Web - Volume Part I*, ISWC'11, pages 585–600, Berlin, Heidelberg, 2011. Springer-Verlag.

[79] A. Schultz, A. Matteini, R. Isele, P. Mendes, C. Bizer, and C. Becker. LDIF - A Framework for Large-Scale Linked Data Integration. In *Proceedings of the 21st International World Wide Web Conference*, WWW '12, 2012.

[80] A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. FedX: Optimization Techniques for Federated Query Processing on Linked Data. In *Proceedings of the 10th International Conference on The Semantic Web - Volume Part I*, ISWC'11, pages 601–616, Berlin, Heidelberg, 2011. Springer-Verlag.

[81] P. Shvaiko and J. Euzenat. Ontology Matching: State of the Art and Future Challenges. *IEEE Trans. Knowl. Data Eng.*, 25(1):158–176, 2013.

[82] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz. Pellet: A Practical OWL-DL Reasoner. *Web Semant.*, 5(2):51–53, June 2007.

[83] G. Skevakis, K. Makris, V. Kalokyri, P. Arapi, and S. Christodoulakis. Metadata Management, Interoperability and Linked Data Publishing Support for Natural History Museums. *International Journal on Digital Libraries*, Apr. 2014.

[84] M. Stollberg, E. Cimpian, A. Mocan, and D. Fensel. A Semantic Web Mediation Architecture. In *Proceedings of the 1st Canadian Semantic Web Working Symposium*, volume 2 of *CSWWS '06*, pages 3–22. Springer, 2006.

[85] H. Stuckenschmidt, R. Vdovjak, G.-J. Houben, and J. Broekstra. Index Structures and Algorithms for Querying Distributed RDF Repositories. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pages 631–639, New York, NY, USA, 2004. ACM.

[86] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: A Large Ontology from Wikipedia and WordNet. *Web Semantics: Science, Services and Agents on the World Wide Web*, 6(3):203–217, 2008.

[87] T. Tran, H. Wang, and P. Haase. Hermes: Data Web Search on a Pay-as-you-go Integration Infrastructure. *Web Semant.*, 7(3):189–203, Sept. 2009.

[88] C. Tsinaraki and S. Christodoulakis. Interoperability of XML Schema Applications with OWL Domain Knowledge and Semantic Web Tools. In *Proceedings of the 2007 OTM Confederated International Conference on On the Move to Meaningful Internet Systems: CoopIS, DOA, ODBASE, GADA, and IS - Volume Part I*, OTM'07, pages 850–869, Berlin, Heidelberg, 2007. Springer-Verlag.

[89] Y. Tzitzikas, N. Spyratos, and P. Constantopoulos. Mediators over Taxonomy-based Information Sources. *The VLDB Journal*, 14(1):112–136, Mar. 2005.

[90] H. Wache, T. Vögele, U. Visser, H. Stuckenschmidt, G. Schuster, H. Neumann, and S. Hübner. Ontology-Based Integration of Information - A Survey of Existing Approaches. In H. Stuckenschmidt, editor, *Proceedings of the 17th International Joint Conference on Artificial Intelligence* , IJCAI '01, pages 108–117, 2001.

[91] G. Wiederhold. Mediators in the Architecture of Future Information Systems. *Computer*, 25(3):38–49, Mar. 1992.

[92] X. Zheng, S. E. Madnick, and X. Li. SPARQL Query Mediation over RDF Data Sources with Disparate Contexts. In *Proceedings of the 5th Linked Data on the Web Workshop*, volume 937 of *LDOW '12*. CEUR-WS.org, 2012.

Appendix A

Semantics Preservation of Query Rewriting: Proofs

This appendix provides the proofs for the semantics preservation of the data triple pattern rewriting rules and functions presented in Section 5.2. For readability reasons, Definition A.1 restates the notion of *semantics preserving rewriting*. Refer to Section 3.3 for the adopted notation and the SPARQL graph pattern semantics, since they are used extensively in the remainder of this appendix. Additionally, note that we consider mappings of type equivalence and we do not provide the proofs for the other mapping types since they follow a similar approach.

Definition A.1 (Semantics Preserving Rewriting). Let G be a mediator ontology schema, let $\bar{S} = \{S_1, \ldots, S_n\}$ be n data source ontology schemas, and let $\bar{M}$ be a complete set of sound GAV mappings between G and $\bar{S}$. Assuming an RDF dataset DS that combines G, $\bar{S}$, $\bar{M}$, along with the respective datasets of G and $\bar{S}$, we state that *the rewriting of a triple pattern t to a graph pattern g, using a mapping $m \in \bar{M}$, is semantics preserving if and only if*:

- Given a variable set $\mathcal{J} = var(t)$, the evaluation of t and g (projected on $\mathcal{J}$) over the RDF dataset DS preserve the exploited mapping's relationship; that is:
  - If mapping m is of type equivalence ($\equiv$), then: $[[t]]_{DS} \equiv \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.
  - If mapping m is of type subsumption ($\sqsubseteq$), then: $[[t]]_{DS} \sqsubseteq \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.
  - If mapping m is of type subsumption ($\sqsupseteq$), then: $[[t]]_{DS} \sqsupseteq \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$. □

In what follows, let $\mathcal{I}$ be the interpretation of the RDF dataset DS (specified in the Definition A.1) consisting of two non-empty sets: (*a*) $\Delta^{\mathcal{I}}$*, the domain of individuals*, and (*b*) $\Delta^{\mathcal{I}}_{\mathcal{D}}$*, the domain of data values*. Moreover, consider an interpretation function which assigns: (*a*) to every class C a set $C^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$, (*b*) to every data range D a set $D^{\mathcal{D}} \subseteq \Delta^{\mathcal{I}}_{\mathcal{D}}$, (*c*) to every object property R a binary relation $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$, (*d*) to every datatype property U a binary relation $U^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}_{\mathcal{D}}$, (*e*) to every individual o an element $o^{\mathcal{I}} \in \Delta^{\mathcal{I}}$, and (*f*) to every data value v an element $v^{\mathcal{I}} = v^{\mathcal{D}}$.

Proof Method Overview. Consider the ontology schemas G, $\bar{S}$, the mapping set $\bar{M}$ and the RDF dataset DS which have been specified in the Definition A.1. Given a triple pattern t consisted of a variable set $\mathcal{J}$, let g be the resulted graph pattern after rewriting t with respect to a mapping $m \in \bar{M}$ of type equivalence for its subject, predicate, or object part.

The proof method starts by showing that every *solution* in the evaluation of t over DS is also a *solution* in the evaluation of g over DS. To this end, we use SPARQL graph pattern semantics along with the semantics of mapping m, exploited by the rewriting process, and we prove that:

$$[[t]]_{DS} \sqsubseteq \pi_{\mathcal{J}}\big([[g]]_{DS}\big) \tag{A.1}$$

Recall that the triple pattern rewriting process preserves the variables of the input triple pattern. Therefore, every variable of t should also appear in g, or in other words, $\mathcal{J}$ is the common variable set between t and g. The method continues by showing that every *solution* in the evaluation of g over the RDF dataset DS is also a *solution* in the evaluation of t over DS for the common variable set $\mathcal{J}$:

$$[[t]]_{DS} \sqsupseteq \pi_{\mathcal{J}}\big([[g]]_{DS}\big) \tag{A.2}$$

From (A.1) and (A.2) we derive that $[[t]]_{DS} \equiv \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$. Since the mapping m, exploited by the rewriting process, is also of type equivalence we conclude the proof. Similarly, for mappings of type subsumption ($\sqsubseteq$, $\sqsupseteq$) we reach either (A.1) or (A.2), proving that the triple pattern rewriting process is semantics preserving; that is, preserves the exploited mapping type semantics.

A.1 Semantics Preservation of Function $\mathcal{D}_s(t, m)$

The function $\mathcal{D}_s(t, m)$ was presented in Table 5.2 and it is used for the rewriting of a data triple pattern t based on a mapping m for its subject part. Following the $\mathcal{D}_s$ function definition, in order to prove that the triple pattern rewriting process preserves the exploited mapping type semantics, it suffices to consider only the case of triple patterns containing an individual in their subject position.

To this end, let o_1, o_2 be individuals, let $t = (o_1, pred, ob)$ be a data triple pattern and let $\mathcal{J}$ be the set of variables appearing in t. Given a mapping m of type $o_1 \equiv o_2$, stating that $o_1^{\mathcal{I}} = o_2^{\mathcal{I}}$, the triple pattern t is reformulated as follows:

$$\mathcal{D}_s(t, m) = (o_2, pred, ob) = g$$

Furthermore, regarding the evaluation of the resulted graph pattern g over the RDF dataset DS

(specified in the Definition A.1) applies that:

$$[[g]]_{DS} = [[(o_2, pred, ob)]]_{DS}$$

Based on the semantics of the input triple pattern t, the mapping type m and the graph pattern g, we consider the following premises:

1. $\forall\, solution \omega \in [[t]]_{DS}\colon\ \exists x, \exists y$, such that $(o_1, x, y) \in DS$. Taking into account that $o_1^{\mathcal{I}} = o_2^{\mathcal{I}}$ we conclude that $(o_2, x, y) \in DS$, and therefore, $\omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.
2. $\forall\, solution\ \omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)\colon\ \exists x, \exists y$, such that $(o_2, x, y) \in DS$. Taking into account that $o_1^{\mathcal{I}} = o_2^{\mathcal{I}}$ we conclude that $(o_1, x, y) \in DS$, and therefore, $\omega \in [[t]]_{DS}$.

From the two premises above, we derive that $[[t]]_{DS} \equiv \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$. This concludes the proof that the function $\mathcal{D}_s$ preserves the exploited mapping type semantics.

A.2 Semantics Preservation of Function $\mathcal{D}_p(t, m)$

The function $\mathcal{D}_p(t, m)$ was presented in Table 5.3 and it is used for the rewriting of a data triple pattern t based on a mapping m for its predicate part. Following the $\mathcal{D}_p$ function definition, in order to prove that the rewriting process preserves the exploited mapping type semantics, it suffices to consider the cases of triple patterns containing either an object or a datatype property in their predicate position.

To begin with, we prove that data triple pattern rewriting based on object property mappings and function $\mathcal{D}_p$ is semantics preserving. To this end, let R_1 be an object property, let $t = (sub, R_1, ob)$ be a data triple pattern and let $\mathcal{J}$ be the set of variables appearing in t. For the different types of object property mappings, we consider the following cases:

1. Given a named object property r_1 and a mapping m of type $R_1 \equiv r_1$, stating that $R_1^{\mathcal{I}} = r_1^{\mathcal{I}}$, the triple pattern t is reformulated as follows:

   $$\mathcal{D}_p(t, m) = (sub, r_1, ob) = g$$

   Furthermore, regarding the evaluation of the resulted graph pattern g over the RDF dataset DS (specified in the Definition A.1) applies that:

   $$[[g]]_{DS} = [[(sub, r_1, ob)]]_{DS}$$

   Based on the semantics of the input triple pattern t, the mapping type m and the graph pattern g, we consider the following premises:

(a) $\forall\, solution\ \omega \in [[t]]_{DS}$: $\exists x, \exists y$, such that $(x, R_1, y) \in DS$, and thus $(x, y) \in R_1^{\mathcal{I}}$. Taking into account that $R_1^{\mathcal{I}} = r_1^{\mathcal{I}}$ we conclude that $(x, y) \in r_1^{\mathcal{I}}$ and $(x, r_1, y) \in DS$. Therefore, $\omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

(b) $\forall\, solution\ \omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$: $\exists x, \exists y$, such that $(x, r_1, y) \in DS$, and thus $(x, y) \in r_1^{\mathcal{I}}$. Taking into account that $R_1^{\mathcal{I}} = r_1^{\mathcal{I}}$ we conclude that $(x, y) \in R_1^{\mathcal{I}}$ and $(x, R_1, y) \in DS$. Therefore, $\omega \in [[t]]_{DS}$.

From the two premises above, we derive that $[[t]]_{DS} \equiv \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

2. Given two object property expressions R_2, R_3 and a mapping m of type $R_1 \equiv R_2 \sqcap R_3$, stating that $R_1^{\mathcal{I}} = R_2^{\mathcal{I}} \cap R_3^{\mathcal{I}}$, the triple pattern t is reformulated as follows:

$$\mathcal{D}_p(t, m) = \mathcal{D}_p\big(t,\ R_1 \rightarrow R_2\big)\ \text{AND}\ \mathcal{D}_p\big(t,\ R_1 \rightarrow R_3\big) = g$$

Furthermore, regarding the evaluation of the resulted graph pattern g over the RDF dataset DS (specified in the Definition A.1) applies that:

$$\begin{aligned}[[g]]_{DS} &= [[\mathcal{D}_p\big(t,\ R_1 \rightarrow R_2\big)\ \text{AND}\ \mathcal{D}_p\big(t,\ R_1 \rightarrow R_3\big)]]_{DS} \\ &= [[\mathcal{D}_p\big(t,\ R_1 \rightarrow R_2\big)]]_{DS} \bowtie [[\mathcal{D}_p\big(t,\ R_1 \rightarrow R_3\big)]]_{DS}\end{aligned}$$

Abusively treating the object property expressions R_2, R_3 as simple object properties, and based on the semantics of the input triple pattern t, the mapping type m and the graph pattern g, we consider the following premises:

(a) $\forall\, solution\ \omega \in [[t]]_{DS}$: $\exists x, \exists y$, such that $(x, R_1, y) \in DS$, and thus $(x, y) \in R_1^{\mathcal{I}}$. Taking into account that $R_1^{\mathcal{I}} = R_2^{\mathcal{I}} \cap R_3^{\mathcal{I}}$ we conclude that $(x, R_2, y) \in DS$ and $(x, R_3, y) \in DS$. Therefore, $\omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

(b) $\forall\, solution\ \omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$: $\exists x, \exists y$, such that $(x, R_2, y), (x, R_3, y) \in DS$, and thus $(x, y) \in R_2^{\mathcal{I}} \cap R_3^{\mathcal{I}}$. Taking into account that $R_1^{\mathcal{I}} = R_2^{\mathcal{I}} \cap R_3^{\mathcal{I}}$ we conclude that $(x, y) \in R_1^{\mathcal{I}}$ and $(x, R_1, y) \in DS$. Therefore, $\omega \in [[t]]_{DS}$.

From the two premises above, we derive that $[[t]]_{DS} \equiv \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

3. Given two object property expressions R_2, R_3 and a mapping m of type $R_1 \equiv R_2 \sqcup R_3$, stating that $R_1^{\mathcal{I}} = R_2^{\mathcal{I}} \cup R_3^{\mathcal{I}}$, the triple pattern t is reformulated as follows:

$$\mathcal{D}_p(t, m) = \mathcal{D}_p\big(t,\ R_1 \rightarrow R_2\big)\ \text{UNION}\ \mathcal{D}_p\big(t,\ R_1 \rightarrow R_3\big) = g$$

Furthermore, regarding the evaluation of the resulted graph pattern g over the RDF dataset

DS (specified in the Definition A.1) applies that:

$$\begin{aligned}[[g]]_{DS} &= [[\mathcal{D}_p\big(t,\ R_1 \rightarrow R_2\big) \text{ UNION } \mathcal{D}_p\big(t,\ R_1 \rightarrow R_3\big)]]_{DS} \\ &= [[\mathcal{D}_p\big(t,\ R_1 \rightarrow R_2\big)]]_{DS} \cup [[\mathcal{D}_p\big(t,\ R_1 \rightarrow R_3\big)]]_{DS}\end{aligned}$$

Abusively treating the object property expressions R_2, R_3 as simple object properties, and based on the semantics of the input triple pattern t, the mapping type m and the graph pattern g, we consider the following premises:

(a) $\forall\, solution \omega \in [[t]]_{DS}$: $\exists x, \exists y$, such that $(x, R_1, y) \in DS$, and thus $(x, y) \in R_1^{\mathcal{I}}$. Taking into account that $R_1^{\mathcal{I}} = R_2^{\mathcal{I}} \cup R_3^{\mathcal{I}}$ we conclude that $(x, R_2, y) \in DS$ or $(x, R_3, y) \in DS$. Therefore, $\omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

(b) $\forall\, solution\ \omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$: $\exists x, \exists y$, such that $(x, R_2, y) \in DS$ or $(x, R_3, y) \in DS$, and thus $(x, y) \in R_2^{\mathcal{I}} \cup R_3^{\mathcal{I}}$. Taking into account that $R_1^{\mathcal{I}} = R_2^{\mathcal{I}} \cup R_3^{\mathcal{I}}$ we conclude that $(x, y) \in R_1^{\mathcal{I}}$ and $(x, R_1, y) \in DS$. Therefore, $\omega \in [[t]]_{DS}$.

From the two premises above, we derive that $[[t]]_{DS} \equiv \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

4. Given two object property expressions R_2, R_3 and a mapping m of type $R_1 \equiv R_2 - R_3$, stating that $R_1^{\mathcal{I}} = R_2^{\mathcal{I}} \setminus R_3^{\mathcal{I}}$, the triple pattern t is reformulated as follows:

$$\mathcal{D}_p(t, m) = \mathcal{D}_p\big(t,\ R_1 \rightarrow R_2\big) \text{ MINUS } \mathcal{D}_p\big(t,\ R_1 \rightarrow R_3\big) = g$$

Furthermore, regarding the evaluation of the resulted graph pattern g over the RDF dataset DS (specified in the Definition A.1) applies that:

$$\begin{aligned}[[g]]_{DS} &= [[\mathcal{D}_p\big(t,\ R_1 \rightarrow R_2\big) \text{ MINUS } \mathcal{D}_p\big(t,\ R_1 \rightarrow R_3\big)]]_{DS} \\ &= [[\mathcal{D}_p\big(t,\ R_1 \rightarrow R_2\big)]]_{DS} \setminus [[\mathcal{D}_p\big(t,\ R_1 \rightarrow R_3\big)]]_{DS}\end{aligned}$$

Abusively treating the object property expressions R_2, R_3 as simple object properties, and based on the semantics of the input triple pattern t, the mapping type m and the graph pattern g, we consider the following premises:

(a) $\forall\, solution\ \omega \in [[t]]_{DS}$: $\exists x, \exists y$, such that $(x, R_1, y) \in DS$, and thus $(x, y) \in R_1^{\mathcal{I}}$. Taking into account that $R_1^{\mathcal{I}} = R_2^{\mathcal{I}} \setminus R_3^{\mathcal{I}}$ we conclude that $(x, R_2, y) \in DS$ and $(x, R_3, y) \notin DS$. Therefore, $\omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

(b) $\forall\, solution \omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$: $\exists x, \exists y$, such that both $(x, R_2, y) \in DS$ and $(x, R_3, y) \notin DS$. Thus, $(x, y) \in R_2^{\mathcal{I}} \setminus R_3^{\mathcal{I}}$. Taking into account that $R_1^{\mathcal{I}} = R_2^{\mathcal{I}} \setminus R_3^{\mathcal{I}}$ we conclude that $(x, y) \in R_1^{\mathcal{I}}$ and $(x, R_1, y) \in DS$. Therefore, $\omega \in [[t]]_{DS}$.

From the two premises above, we derive that $[[t]]_{DS} \equiv \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

5. Given two object property expressions R_2, R_3 and a mapping m of type $R_1 \equiv R_2 \circ R_3$, stating that $R_1^{\mathcal{I}} = \mathcal{L} = \{(\alpha, c) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid \exists b.\ (\alpha, b) \in R_2^{\mathcal{I}} \wedge (b, c) \in R_3^{\mathcal{I}}\}$, the triple pattern t is reformulated as follows:

$$\mathcal{D}_p(t, m) = \mathcal{D}_p\big((sub, R, ?v),\ R \rightarrow R_2\big) \text{ AND } \mathcal{D}_p\big((?v, R, ob),\ R \rightarrow R_3\big) = g$$

Furthermore, regarding the evaluation of the resulted graph pattern g over the RDF dataset DS (specified in the Definition A.1) applies that:

$$\begin{aligned}[[g]]_{DS} &= [[\mathcal{D}_p\big((sub, R, ?v),\ R \rightarrow R_2\big) \text{ AND } \mathcal{D}_p\big((?v, R, ob),\ R \rightarrow R_3\big)]]_{DS} \\ &= [[\mathcal{D}_p\big((sub, R, ?v),\ R \rightarrow R_2\big)]]_{DS} \bowtie [[\mathcal{D}_p\big((?v, R, ob),\ R \rightarrow R_3\big)]]_{DS}\end{aligned}$$

Abusively treating the object property expressions R_2, R_3 as simple object properties, and based on the semantics of the input triple pattern t, the mapping type m and the graph pattern g, we consider the following premises:

(a) $\forall\, solution\ \omega \in [[t]]_{DS}$: $\exists x, \exists y$, such that $(x, R_1, y) \in DS$, and thus $(x, y) \in R_1^{\mathcal{I}}$. Taking into account that $R_1^{\mathcal{I}} = \mathcal{L}$ we conclude that $\exists z$ such that $(x, R_2, z) \in DS$ and $(z, R_3, y) \in DS$. Therefore, $\omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

(b) $\forall\, solution\ \omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$: $\exists x, \exists y, \exists z$, so that $(x, R_2, z), (z, R_3, y) \in DS$, and thus $(x, y) \in \mathcal{L}$. Taking into account that $R_1^{\mathcal{I}} = \mathcal{L}$ we conclude that $(x, y) \in R_1^{\mathcal{I}}$ and $(x, R_1, y) \in DS$. Therefore, $\omega \in [[t]]_{DS}$.

From the two premises above, we derive that $[[t]]_{DS} \equiv \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

6. Given an object property expression R_2 and a mapping m of type $R_1 \equiv R_2^{-}$, stating that $R_1^{\mathcal{I}} = \mathcal{L} = \{(b, \alpha) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid (\alpha, b) \in R_2^{\mathcal{I}}\}$, the triple pattern t is reformulated as follows:

$$\mathcal{D}_p(t, m) = \mathcal{D}_p\big((ob, R, sub),\ R \rightarrow R_2\big) = g$$

Furthermore, regarding the evaluation of the resulted graph pattern g over the RDF dataset DS (specified in the Definition A.1) applies that:

$$[[g]]_{DS} = [[\mathcal{D}_p\big((ob, R, sub),\ R \rightarrow R_2\big)]]_{DS}$$

Abusively treating the object property expression R_2 as a simple object property, and based on the semantics of the input triple pattern t, the mapping type m and the graph pattern g, we consider the following premises:

(a) $\forall\, solution\ \omega \in [[t]]_{DS}$: $\exists x, \exists y$, such that $(x, R_1, y) \in DS$, and thus $(x, y) \in R_1^{\mathcal{I}}$. Taking into account that $R_1^{\mathcal{I}} = \mathcal{L}$ we conclude that $(y, R_2, x) \in DS$. Therefore,

$\omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

(b) $\forall\, solution\, \omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$: $\exists x, \exists y$, such that $(x, R_2, y) \in DS$, and thus $(x, y) \in \mathcal{L}$. Taking into account that $R_1^{\mathcal{I}} = \mathcal{L}$ we conclude that $(y, x) \in R_1^{\mathcal{I}}$ and $(y, R_1, x) \in DS$. Therefore, $\omega \in [[t]]_{DS}$.

From the two premises above, we derive that $[[t]]_{DS} \equiv \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

7. Given an object property expression R_2 and a mapping m of type $R_1 \equiv R_2^+$, stating that $R_1^{\mathcal{I}} = \mathcal{L} = \bigcup_{n \geq 1}(R_2^{\mathcal{I}})^n$, the triple pattern t is reformulated as follows:

$$\mathcal{D}_p(t, m) = \mathcal{D}_p\big(t,\ R_1 \rightarrow R_2\big) \text{ UNION } \mathcal{D}_p\big(t,\ R_1 \rightarrow R_2 \circ R_2\big) = g$$

Furthermore, regarding the evaluation of the resulted graph pattern g over the RDF dataset DS (specified in the Definition A.1) applies that:

$$\begin{aligned}[[g]]_{DS} &= [[\mathcal{D}_p\big(t,\ R_1 \rightarrow R_2\big) \text{ UNION } \mathcal{D}_p\big(t,\ R_1 \rightarrow R_2 \circ R_2\big)]]_{DS} \\ &= [[\mathcal{D}_p\big(t,\ R_1 \rightarrow R_2\big)]]_{DS} \cup [[\mathcal{D}_p\big(t,\ R_1 \rightarrow R_2 \circ R_2\big)]]_{DS}\end{aligned}$$

Abusively treating the object property expression R_2 as a simple object property, and based on the semantics of the input triple pattern t, the mapping type m and the graph pattern g, we consider the following premises:

(a) $\forall\, solution\, \omega \in [[t]]_{DS}$: $\exists x, \exists y$, such that $(x, R_1, y) \in DS$, and thus $(x, y) \in R_1^{\mathcal{I}}$. Taking into account that $R_1^{\mathcal{I}} = \mathcal{L}$ we conclude that $(x, R_2, y) \in DS$. Therefore, $\omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

(b) $\forall\, solution\, \omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$: $\exists x, \exists y$, such that $(x, R_2, y) \in DS$, or $\exists z$ so that $(x, R_2, z), (z, R_2, y) \in DS$. Thus, $(x, y) \in \mathcal{L}$. Taking into account that $R_1^{\mathcal{I}} = \mathcal{L}$ we conclude that $(x, y) \in R_1^{\mathcal{I}}$ and $(x, R_1, y) \in DS$. Therefore, $\omega \in [[t]]_{DS}$.

From the two premises above, we derive that $[[t]]_{DS} \equiv \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

8. Given two object property expressions R_2 and R_3, as well as a mapping m of type $R_1 \equiv \exists(R_2)(R_3).P_1$, stating that $R_1^{\mathcal{I}} = \mathcal{L} = \{(\alpha, b) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid \exists c_1, c_2.\, (\alpha, c_1) \in R_2^{\mathcal{I}} \wedge (b, c_2) \in R_3^{\mathcal{I}} \wedge (c_1, c_2) \in P_1^{\mathcal{D}}\}$, the triple pattern t is reformulated as follows:

$$\begin{aligned}\mathcal{D}_p(t, m) = \mathcal{D}_p\big((sub, R, ?v_1),\ R \rightarrow R_2\big) \text{ AND } \mathcal{D}_p\big((ob, R, ?v_2),\ R \rightarrow R_3\big) \\ \text{FILTER } \mathcal{T}_b\big(P_1, ?v_1, ?v_2\big) = g\end{aligned}$$

Furthermore, regarding the evaluation of the resulted graph pattern g over the RDF dataset

DS (specified in the Definition A.1) applies that:

$$\begin{aligned}[[g]]_{DS} &= [[\mathcal{D}_p\big((sub, R, ?v_1),\ R \rightarrow R_2\big) \text{ AND } \mathcal{D}_p\big((ob, R, ?v_2),\ R \rightarrow R_3\big) \\ &\quad \text{FILTER } \mathcal{T}_b\big(P_1, ?v_1, ?v_2\big)]]_{DS} \\ &= \{\omega \in [[\mathcal{D}_p\big((sub, R, ?v_1),\ R \rightarrow R_2\big)]]_{DS} \bowtie [[\mathcal{D}_p\big((ob, R, ?v_2), \\ &\quad R \rightarrow R_3\big)]]_{DS} \mid \omega \models \mathcal{T}_b\big(P_1, ?v_1, ?v_2\big)\}\end{aligned}$$

Abusively treating the object property expressions R_2, R_3 as simple object properties, and based on the semantics of the input triple pattern t, the mapping type m and the graph pattern g, we consider the following premises:

(a) $\forall\, solution\omega \in [[t]]_{DS}$: $\exists x, \exists y$, such that $(x, R_1, y) \in DS$, and thus $(x, y) \in R_1^{\mathcal{I}}$. Taking into account that $R_1^{\mathcal{I}} = \mathcal{L}$ we conclude that $\exists z_1, \exists z_2$ so that $(x, R_2, z_1), (y, R_3, z_2) \in DS$ and $(z_1, z_2) \in P_1^{\mathcal{D}}$. Therefore, $\omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

(b) $\forall\, solution\ \omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$: $\exists x, \exists y, \exists z_1, \exists z_2$, so that $(x, R_2, z_1) \in DS$, $(y, R_3, z_2) \in DS$ and $(z_1, z_2) \in P_1^{\mathcal{D}}$. Thus, $(x, y) \in \mathcal{L}$. Taking into account that $R_1^{\mathcal{I}} = \mathcal{L}$ we conclude that $(x, y) \in R_1^{\mathcal{I}}$ and $(x, R_1, y) \in DS$. Therefore, $\omega \in [[t]]_{DS}$.

From the two premises above, we derive that $[[t]]_{DS} \equiv \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

9. Given two datatype property expressions U_1, U_2 as well as a mapping m of type $R_1 \equiv \exists(U_1)(U_2).P_1$, stating that $R_1^{\mathcal{I}} = \mathcal{L} = \{(\alpha, b) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid \exists c_1, c_2.\, (\alpha, c_1) \in U_1^{\mathcal{I}} \wedge (b, c_2) \in U_2^{\mathcal{I}} \wedge (c_1, c_2) \in P_1^{\mathcal{D}}\}$, the triple pattern t is reformulated as follows:

$$\begin{aligned}\mathcal{D}_p(t, m) = \mathcal{D}_p\big((sub, U, ?v_1),\ U \rightarrow U_1\big) &\text{ AND } \mathcal{D}_p\big((ob, U, ?v_2),\ U \rightarrow U_2\big) \\ \text{FILTER } \mathcal{T}_b\big(P_1, ?v_1, ?v_2\big) &= g\end{aligned}$$

Furthermore, regarding the evaluation of the resulted graph pattern g over the RDF dataset DS (specified in the Definition A.1) applies that:

$$\begin{aligned}[[g]]_{DS} &= [[\mathcal{D}_p\big((sub, U, ?v_1),\ U \rightarrow U_1\big) \text{ AND } \mathcal{D}_p\big((ob, U, ?v_2),\ U \rightarrow U_2\big) \\ &\quad \text{FILTER } \mathcal{T}_b\big(P_1, ?v_1, ?v_2\big)]]_{DS} \\ &= \{\omega \in [[\mathcal{D}_p\big((sub, U, ?v_1),\ U \rightarrow U_1\big)]]_{DS} \bowtie [[\mathcal{D}_p\big((ob, U, ?v_2), \\ &\quad U \rightarrow U_2\big)]]_{DS} \mid \omega \models \mathcal{T}_b\big(P_1, ?v_1, ?v_2\big)\}\end{aligned}$$

Abusively treating the datatype property expressions U_1, U_2 as simple datatype properties, and based on the semantics of the input triple pattern t, the mapping type m and the graph pattern g, we consider the following premises:

(a) $\forall\, solution\omega \in [[t]]_{DS}$: $\exists x, \exists y$, such that $(x, R_1, y) \in DS$, and thus $(x, y) \in R_1^{\mathcal{I}}$. Tak-

ing into account that $R_1^{\mathcal{I}} = \mathcal{L}$ we conclude that $\exists z_1, \exists z_2$ so that $(x, U_1, z_1), (y, U_2, z_2) \in DS$ and $(z_1, z_2) \in P_1^{\mathcal{D}}$. Therefore, $\omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

(b) $\forall\, solution \omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)\colon\ \exists x, \exists y, \exists z_1, \exists z_2$, such that $(x, U_1, z_1) \in DS$, $(y, U_2, z_2) \in DS$ and $(z_1, z_2) \in P_1^{\mathcal{D}}$. Thus, $(x, y) \in \mathcal{L}$. Taking into account that $R_1^{\mathcal{I}} = \mathcal{L}$ we conclude that $(x, y) \in R_1^{\mathcal{I}}$ and $(x, R_1, y) \in DS$. Therefore, $\omega \in [[t]]_{DS}$.

From the two premises above, we derive that $[[t]]_{DS} \equiv \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

10. Given an object property expression R_2, a class expression C_1 and a mapping m of type $R_1 \equiv R_2 \upharpoonright C_1$, stating that $R_1^{\mathcal{I}} = \mathcal{L} = \{(\alpha, b) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid (\alpha, b) \in R_2^{\mathcal{I}} \wedge \alpha \in C_1^{\mathcal{I}}\}$, the triple pattern t is reformulated as follows:

$$\mathcal{D}_p(t, m) = \mathcal{D}_p\big(t,\ R_1 \rightarrow R_2\big) \text{ AND } \mathcal{D}_o\big((sub, rdf{:}type, C),\ C \rightarrow C_1\big) = g$$

Furthermore, regarding the evaluation of the resulted graph pattern g over the RDF dataset DS (specified in the Definition A.1) applies that:

$$\begin{aligned}[[g]]_{DS} &= [[\mathcal{D}_p\big(t,\ R_1 \rightarrow R_2\big) \text{ AND } \mathcal{D}_o\big((sub, rdf{:}type, C),\ C \rightarrow C_1\big)]]_{DS} \\ &= [[\mathcal{D}_p\big(t,\ R_1 \rightarrow R_2\big)]]_{DS} \bowtie [[\mathcal{D}_o\big((sub, rdf{:}type, C),\ C \rightarrow C_1\big)]]_{DS}\end{aligned}$$

Abusively treating the expressions R_2, C_1 as simple property and class respectively, and based on the semantics of the input triple pattern t, the mapping type m and the graph pattern g, we consider the following premises:

(a) $\forall\, solution \omega \in [[t]]_{DS}\colon\ \exists x, \exists y$, such that $(x, R_1, y) \in DS$, and thus $(x, y) \in R_1^{\mathcal{I}}$. Taking into account that $R_1^{\mathcal{I}} = \mathcal{L}$ we conclude that $(x, R_2, y) \in DS$ and $(x, rdf{:}type, C_1) \in DS$. Therefore, $\omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

(b) $\forall\, solution\, \omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)\colon\ \exists x, \exists y$, so that $(x, R_2, y), (x, rdf{:}type, C_1) \in DS$, and thus $(x, y) \in \mathcal{L}$. Taking into account that $R_1^{\mathcal{I}} = \mathcal{L}$ we conclude that $(x, y) \in R_1^{\mathcal{I}}$ and $(x, R_1, y) \in DS$. Therefore, $\omega \in [[t]]_{DS}$.

From the two premises above, we derive that $[[t]]_{DS} \equiv \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

11. Given an object property expression R_2, a class expression C_1 and a mapping m of type $R_1 \equiv R_2 \downharpoonright C_1$, stating that $R_1^{\mathcal{I}} = \mathcal{L} = \{(\alpha, b) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid (\alpha, b) \in R_2^{\mathcal{I}} \wedge b \in C_1^{\mathcal{I}}\}$, the triple pattern t is reformulated as follows:

$$\mathcal{D}_p(t, m) = \mathcal{D}_p\big(t,\ R_1 \rightarrow R_2\big) \text{ AND } \mathcal{D}_o\big((ob, rdf{:}type, C),\ C \rightarrow C_1\big) = g$$

Furthermore, regarding the evaluation of the resulted graph pattern g over the RDF dataset

DS (specified in the Definition A.1) applies that:

$$\begin{aligned}[[g]]_{DS} &= [[\mathcal{D}_p\big(t,\ R_1 \rightarrow R_2\big) \text{ AND } \mathcal{D}_o\big((ob, rdf{:}type, C),\ C \rightarrow C_1\big)]]_{DS} \\ &= [[\mathcal{D}_p\big(t,\ R_1 \rightarrow R_2\big)]]_{DS} \bowtie [[\mathcal{D}_o\big((ob, rdf{:}type, C),\ C \rightarrow C_1\big)]]_{DS}\end{aligned}$$

Abusively treating the expressions R_2, C_1 as simple property and class respectively, and based on the semantics of the input triple pattern t, the mapping type m and the graph pattern g, we consider the following premises:

(a) $\forall\, solution \omega \in [[t]]_{DS}$: $\exists x, \exists y$, such that $(x, R_1, y) \in DS$, and thus $(x, y) \in R_1^{\mathcal{I}}$. Taking into account that $R_1^{\mathcal{I}} = \mathcal{L}$ we conclude that $(x, R_2, y) \in DS$ and $(y, rdf{:}type, C_1) \in DS$. Therefore, $\omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

(b) $\forall\, solution\ \omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$: $\exists x, \exists y$, so that $(x, R_2, y), (y, rdf{:}type, C_1) \in DS$, and thus $(x, y) \in \mathcal{L}$. Taking into account that $R_1^{\mathcal{I}} = \mathcal{L}$ we conclude that $(x, y) \in R_1^{\mathcal{I}}$ and $(x, R_1, y) \in DS$. Therefore, $\omega \in [[t]]_{DS}$.

From the two premises above, we derive that $[[t]]_{DS} \equiv \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

Similarly, we prove that data triple pattern rewriting using the function $\mathcal{D}_p$ along with a datatype property mapping is semantics preserving. To this end, let U_1 be a datatype property, let $t = (sub, U_1, ob)$ be a data triple pattern and let $\mathcal{J}$ be the set of variables appearing in t. For the different types of datatype property mappings, we consider the following cases:

1. Given a named datatype property u_1 and a mapping m of type $U_1 \equiv u_1$, stating that $U_1^{\mathcal{I}} = u_1^{\mathcal{I}}$, the triple pattern t is reformulated as follows:

$$\mathcal{D}_p(t, m) = (sub, u_1, ob) = g$$

Furthermore, regarding the evaluation of the resulted graph pattern g over the RDF dataset DS (specified in the Definition A.1) applies that:

$$[[g]]_{DS} = [[(sub, u_1, ob)]]_{DS}$$

Based on the semantics of the input triple pattern t, the mapping type m and the graph pattern g, we consider the following premises:

(a) $\forall\, solution\ \omega \in [[t]]_{DS}$: $\exists x, \exists y$, such that $(x, U_1, y) \in DS$, and thus $(x, y) \in U_1^{\mathcal{I}}$. Taking into account that $U_1^{\mathcal{I}} = u_1^{\mathcal{I}}$ we conclude that $(x, y) \in u_1^{\mathcal{I}}$ and $(x, u_1, y) \in DS$. Therefore, $\omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

(b) $\forall\, solution \omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$: $\exists x, \exists y$, such that $(x, u_1, y) \in DS$, and thus $(x, y) \in u_1^{\mathcal{I}}$. Taking into account that $U_1^{\mathcal{I}} = u_1^{\mathcal{I}}$ we conclude that $(x, y) \in U_1^{\mathcal{I}}$ and $(x, U_1, y) \in DS$.

Therefore, $\omega \in [[t]]_{DS}$.

From the two premises above, we derive that $[[t]]_{DS} \equiv \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

2. Given two datatype property expressions U_2, U_3 and a mapping m of type $U_1 \equiv U_2 \sqcap U_3$, stating that $U_1^{\mathcal{I}} = U_2^{\mathcal{I}} \cap U_3^{\mathcal{I}}$, the triple pattern t is reformulated as follows:

$$\mathcal{D}_p(t, m) = \mathcal{D}_p\big(t,\ U_1 \rightarrow U_2\big) \text{ AND } \mathcal{D}_p\big(t,\ U_1 \rightarrow U_3\big) = g$$

Furthermore, regarding the evaluation of the resulted graph pattern g over the RDF dataset DS (specified in the Definition A.1) applies that:

$$\begin{aligned}[[g]]_{DS} &= [[\mathcal{D}_p\big(t,\ U_1 \rightarrow U_2\big) \text{ AND } \mathcal{D}_p\big(t,\ U_1 \rightarrow U_3\big)]]_{DS} \\ &= [[\mathcal{D}_p\big(t,\ U_1 \rightarrow U_2\big)]]_{DS} \bowtie [[\mathcal{D}_p\big(t,\ U_1 \rightarrow U_3\big)]]_{DS}\end{aligned}$$

Abusively treating the datatype property expressions U_2, U_3 as simple datatype properties, and based on the semantics of the input triple pattern t, the mapping type m and the graph pattern g, we consider the following premises:

(a) $\forall\, solution\ \omega \in [[t]]_{DS}$: $\exists x, \exists y$, such that $(x, U_1, y) \in DS$, and thus $(x, y) \in U_1^{\mathcal{I}}$. Taking into account that $U_1^{\mathcal{I}} = U_2^{\mathcal{I}} \cap U_3^{\mathcal{I}}$ we conclude that $(x, U_2, y) \in DS$ and $(x, U_3, y) \in DS$. Therefore, $\omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

(b) $\forall\, solution\ \omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$: $\exists x, \exists y$, such that $(x, U_2, y), (x, U_3, y) \in DS$, and thus $(x, y) \in U_2^{\mathcal{I}} \cap U_3^{\mathcal{I}}$. Taking into account that $U_1^{\mathcal{I}} = U_2^{\mathcal{I}} \cap U_3^{\mathcal{I}}$ we conclude that $(x, y) \in U_1^{\mathcal{I}}$ and $(x, U_1, y) \in DS$. Therefore, $\omega \in [[t]]_{DS}$.

From the two premises above, we derive that $[[t]]_{DS} \equiv \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

3. Given two datatype property expressions U_2, U_3 and a mapping m of type $U_1 \equiv U_2 \sqcup U_3$, stating that $U_1^{\mathcal{I}} = U_2^{\mathcal{I}} \cup U_3^{\mathcal{I}}$, the triple pattern t is reformulated as follows:

$$\mathcal{D}_p(t, m) = \mathcal{D}_p\big(t,\ U_1 \rightarrow U_2\big) \text{ UNION } \mathcal{D}_p\big(t,\ U_1 \rightarrow U_3\big) = g$$

Furthermore, regarding the evaluation of the resulted graph pattern g over the RDF dataset DS (specified in the Definition A.1) applies that:

$$\begin{aligned}[[g]]_{DS} &= [[\mathcal{D}_p\big(t,\ U_1 \rightarrow U_2\big) \text{ UNION } \mathcal{D}_p\big(t,\ U_1 \rightarrow U_3\big)]]_{DS} \\ &= [[\mathcal{D}_p\big(t,\ U_1 \rightarrow U_2\big)]]_{DS} \cup [[\mathcal{D}_p\big(t,\ U_1 \rightarrow U_3\big)]]_{DS}\end{aligned}$$

Abusively treating the datatype property expressions U_2, U_3 as simple datatype properties, and based on the semantics of the input triple pattern t, the mapping type m and the graph pattern g, we consider the following premises:

(a) $\forall\, solution \omega \in [[t]]_{DS}$: $\exists x, \exists y$, such that $(x, U_1, y) \in DS$, and thus $(x, y) \in U_1^{\mathcal{I}}$. Taking into account that $U_1^{\mathcal{I}} = U_2^{\mathcal{I}} \cup U_3^{\mathcal{I}}$ we conclude that $(x, U_2, y) \in DS$ or $(x, U_3, y) \in DS$. Therefore, $\omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

(b) $\forall\, solution\, \omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$: $\exists x, \exists y$, such that $(x, U_2, y) \in DS$ or $(x, U_3, y) \in DS$, and thus $(x, y) \in U_2^{\mathcal{I}} \cup U_3^{\mathcal{I}}$. Taking into account that $U_1^{\mathcal{I}} = U_2^{\mathcal{I}} \cup U_3^{\mathcal{I}}$ we conclude that $(x, y) \in U_1^{\mathcal{I}}$ and $(x, U_1, y) \in DS$. Therefore, $\omega \in [[t]]_{DS}$.

From the two premises above, we derive that $[[t]]_{DS} \equiv \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

4. Given two datatype property expressions U_2, U_3 and a mapping m of type $U_1 \equiv U_2 - U_3$, stating that $U_1^{\mathcal{I}} = U_2^{\mathcal{I}} \setminus U_3^{\mathcal{I}}$, the triple pattern t is reformulated as follows:

$$\mathcal{D}_p(t, m) = \mathcal{D}_p\big(t,\ U_1 \to U_2\big) \text{ MINUS } \mathcal{D}_p\big(t,\ U_1 \to U_3\big) = g$$

Furthermore, regarding the evaluation of the resulted graph pattern g over the RDF dataset DS (specified in the Definition A.1) applies that:

$$\begin{aligned}[[g]]_{DS} &= [[\mathcal{D}_p\big(t,\ U_1 \to U_2\big) \text{ MINUS } \mathcal{D}_p\big(t,\ U_1 \to U_3\big)]]_{DS} \\ &= [[\mathcal{D}_p\big(t,\ U_1 \to U_2\big)]]_{DS} \setminus [[\mathcal{D}_p\big(t,\ U_1 \to U_3\big)]]_{DS}\end{aligned}$$

Abusively treating the datatype property expressions U_2, U_3 as simple datatype properties, and based on the semantics of the input triple pattern t, the mapping type m and the graph pattern g, we consider the following premises:

(a) $\forall\, solution\ \omega \in [[t]]_{DS}$: $\exists x, \exists y$, such that $(x, U_1, y) \in DS$, and thus $(x, y) \in U_1^{\mathcal{I}}$. Taking into account that $U_1^{\mathcal{I}} = U_2^{\mathcal{I}} \setminus U_3^{\mathcal{I}}$ we conclude that $(x, U_2, y) \in DS$ and $(x, U_3, y) \notin DS$. Therefore, $\omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

(b) $\forall\, solution\, \omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$: $\exists x, \exists y$, such that $(x, U_2, y) \in DS$ and $(x, U_3, y) \notin D$. Thus $(x, y) \in U_2^{\mathcal{I}} \setminus U_3^{\mathcal{I}}$. Taking into account that $U_1^{\mathcal{I}} = U_2^{\mathcal{I}} \setminus U_3^{\mathcal{I}}$ we conclude that $(x, y) \in U_1^{\mathcal{I}}$ and $(x, U_1, y) \in DS$. Therefore, $\omega \in [[t]]_{DS}$.

From the two premises above, we derive that $[[t]]_{DS} \equiv \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

5. Given an object property expression R_1, a datatype property expression U_2 and a mapping m of type $U_1 \equiv R_1 \circ U_2$, stating that $U_1^{\mathcal{I}} = \mathcal{L} = \{(\alpha, c) \in \Delta^{\mathcal{I}} \times \Delta_{\mathcal{D}}^{\mathcal{I}} \mid \exists b.(\alpha, b) \in R_1^{\mathcal{I}} \wedge (b, c) \in U_2^{\mathcal{I}}\}$, the triple pattern t is reformulated as follows:

$$\mathcal{D}_p(t, m) = \mathcal{D}_p\big((sub, R, ?v),\ R \to R_1\big) \text{ AND } \mathcal{D}_p\big((?v, U, ob),\ U \to U_2\big) = g$$

Furthermore, regarding the evaluation of the resulted graph pattern g over the RDF dataset

DS (specified in the Definition A.1) applies that:

$$[[g]]_{DS} = [[\mathcal{D}_p\big((sub, R, ?v),\ R \rightarrow R_1\big) \text{ AND } \mathcal{D}_p\big((?v, U, ob),\ U \rightarrow U_2\big)]]_{DS}$$
$$= [[\mathcal{D}_p\big((sub, R, ?v),\ R \rightarrow R_1\big)]]_{DS} \bowtie [[\mathcal{D}_p\big((?v, U, ob),\ U \rightarrow U_2\big)]]_{DS}$$

Abusively treating the property expressions R_1, U_2 as simple properties, and based on the semantics of the input triple pattern t, the mapping type m and the graph pattern g, we consider the following premises:

(a) $\forall\, solution\ \omega \in [[t]]_{DS}$: $\exists x, \exists y$, such that $(x, U_1, y) \in DS$, and thus $(x, y) \in U_1^{\mathcal{I}}$. Taking into account that $U_1^{\mathcal{I}} = \mathcal{L}$ we conclude that $\exists z$ such that $(x, R_1, z) \in DS$ and $(z, U_2, y) \in DS$. Therefore, $\omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

(b) $\forall\, solution\ \omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$: $\exists x, \exists y, \exists z$, so that $(x, R_1, z), (z, U_2, y) \in DS$, and thus $(x, y) \in \mathcal{L}$. Taking into account that $U_1^{\mathcal{I}} = \mathcal{L}$ we conclude that $(x, y) \in U_1^{\mathcal{I}}$ and $(x, U_1, y) \in DS$. Therefore, $\omega \in [[t]]_{DS}$.

From the two premises above, we derive that $[[t]]_{DS} \equiv \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

6. Given a datatype property expression U_2, a class expression C_1 and a mapping m of type $U_1 \equiv U_2 \restriction C_1$, stating that $U_1^{\mathcal{I}} = \mathcal{L} = \{(\alpha, b) \in \Delta^{\mathcal{I}} \times \Delta_{\mathcal{D}}^{\mathcal{I}} \mid (\alpha, b) \in U_2^{\mathcal{I}} \wedge \alpha \in C_1^{\mathcal{I}}\}$, the triple pattern t is reformulated as follows:

$$\mathcal{D}_p(t, m) = \mathcal{D}_p\big(t,\ U_1 \rightarrow U_2\big) \text{ AND } \mathcal{D}_o\big((sub, rdf{:}type, C),\ C \rightarrow C_1\big) = g$$

Furthermore, regarding the evaluation of the resulted graph pattern g over the RDF dataset DS (specified in the Definition A.1) applies that:

$$[[g]]_{DS} = [[\mathcal{D}_p\big(t,\ U_1 \rightarrow U_2\big) \text{ AND } \mathcal{D}_o\big((sub, rdf{:}type, C),\ C \rightarrow C_1\big)]]_{DS}$$
$$= [[\mathcal{D}_p\big(t,\ U_1 \rightarrow U_2\big)]]_{DS} \bowtie [[\mathcal{D}_o\big((sub, rdf{:}type, C),\ C \rightarrow C_1\big)]]_{DS}$$

Abusively treating the expressions U_2, C_1 as simple property and class respectively, and based on the semantics of the input triple pattern t, the mapping type m and the graph pattern g, we consider the following premises:

(a) $\forall\, solution \omega \in [[t]]_{DS}$: $\exists x, \exists y$, such that $(x, U_1, y) \in DS$, and thus $(x, y) \in U_1^{\mathcal{I}}$. Taking into account that $U_1^{\mathcal{I}} = \mathcal{L}$ we conclude that $(x, U_2, y) \in DS$ and $(x, rdf{:}type, C_1) \in DS$. Therefore, $\omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

(b) $\forall\, solution\ \omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$: $\exists x, \exists y$, so that $(x, U_2, y), (x, rdf{:}type, C_1) \in DS$, and thus $(x, y) \in \mathcal{L}$. Taking into account that $U_1^{\mathcal{I}} = \mathcal{L}$ we conclude that $(x, y) \in U_1^{\mathcal{I}}$ and $(x, U_1, y) \in DS$. Therefore, $\omega \in [[t]]_{DS}$.

From the two premises above, we derive that $[[t]]_{DS} \equiv \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

7. Given a datatype property expression U_2, a data range D_1 and a mapping m of type $U_1 \equiv U_2 \restriction D_1$, stating that $U_1^{\mathcal{I}} = \mathcal{L} = \{(\alpha, b) \in \Delta^{\mathcal{I}} \times \Delta_{\mathcal{D}}^{\mathcal{I}} \mid (\alpha, b) \in U_2^{\mathcal{I}} \wedge b \in D_1^{\mathcal{D}}\}$, the triple pattern t is reformulated as follows:

$$\mathcal{D}_p(t, m) = \mathcal{D}_p\big(t,\ U_1 \rightarrow U_2\big) \text{ FILTER } \mathcal{T}_d\big(D_1, ob\big) = g$$

Furthermore, regarding the evaluation of the resulted graph pattern g over the RDF dataset DS (specified in the Definition A.1) applies that:

$$\begin{aligned}[[g]]_{DS} &= [[\mathcal{D}_p\big(t,\ U_1 \rightarrow U_2\big) \text{ FILTER } \mathcal{T}_d\big(D_1, ob\big)]]_{DS} \\ &= \{\omega \in [[\mathcal{D}_p\big(t,\ U_1 \rightarrow U_2\big)]]_{DS} \mid \omega \models \mathcal{T}_d\big(D_1, ob\big)\}\end{aligned}$$

Abusively treating the datatype property expression U_2 as a simple datatype property respectively, and based on the semantics of the input triple pattern t, the mapping type m and the graph pattern g, we consider the following premises:

(a) $\forall\, solution\ \omega \in [[t]]_{DS}$: $\exists x, \exists y$, such that $(x, U_1, y) \in DS$, and thus $(x, y) \in U_1^{\mathcal{I}}$. Taking into account that $U_1^{\mathcal{I}} = \mathcal{L}$ we conclude that $(x, U_2, y) \in DS$ and $y \in D_1^{\mathcal{D}}$. Therefore, $\omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

(b) $\forall\, solution\ \omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$: $\exists x$ and $\exists y \in D_1^{\mathcal{D}}$, such that $(x, U_2, y) \in DS$, and thus $(x, y) \in \mathcal{L}$. Taking into account that $U_1^{\mathcal{I}} = \mathcal{L}$ we conclude that $(x, y) \in U_1^{\mathcal{I}}$ and $(x, U_1, y) \in DS$. Therefore, $\omega \in [[t]]_{DS}$.

From the two premises above, we derive that $[[t]]_{DS} \equiv \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

8. Given a datatype property expression U_2 along with a mapping m of type $U_1 \equiv trans(U_2)$, stating that $U_1^{\mathcal{I}} = trans(U_2^{\mathcal{I}})$, the triple pattern t is reformulated as follows:

$$\begin{aligned}\mathcal{D}_p(t, m) = \text{SELECT } &\big(sub,\ trans(?v) \text{ AS } ob\big) \\ &\text{WHERE } \big(\mathcal{D}_p\big((sub, U, ?v),\ U \rightarrow U_2\big)\big) = g\end{aligned}$$

Furthermore, regarding the evaluation of the resulted graph pattern g over the RDF dataset DS (specified in the Definition A.1) applies that:

$$\begin{aligned}[[g]]_{DS} &= [[\text{SELECT } \big(sub,\ trans(?v) \text{ AS } ob\big) \\ &\qquad \text{WHERE } \big(\mathcal{D}_p\big((sub, U, ?v),\ U \rightarrow U_2\big)\big)]]_{DS} \\ &= \pi_{sub,\ ob:\, trans(?v)}[[\mathcal{D}_p\big((sub, U, ?v),\ U \rightarrow U_2\big)]]_{DS}\end{aligned}$$

Abusively treating the datatype property expressions U_2, U_3 as simple datatype properties,

and based on the semantics of the input triple pattern t, the mapping type m and the graph pattern g, we consider the following premises:

(a) $\forall\, solution\ \omega \in [[t]]_{DS}$: $\exists x, \exists y$, such that $(x, U_1, y) \in DS$, and thus $(x, y) \in U_1^{\mathcal{I}}$. Taking into account that $U_1^{\mathcal{I}} = trans(U_2^{\mathcal{I}})$ we conclude that $\exists z$ such that $(x, U_2, z) \in DS$ and $y = trans(z)$. Therefore, $\omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

(b) $\forall\, solution\ \omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$: $\exists x, \exists y, \exists z$, so that $(x, U_2, z) \in DS$ and $y = trans(z)$. Thus, $(x, y) \in trans(U_2^{\mathcal{I}})$. Taking into account that $U_1^{\mathcal{I}} = trans(U_2^{\mathcal{I}})$ we conclude that $(x, y) \in U_1^{\mathcal{I}}$ and $(x, U_1, y) \in DS$. Therefore, $\omega \in [[t]]_{DS}$.

From the two premises above, we derive that $[[t]]_{DS} \equiv \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

This concludes the proof that the function $\mathcal{D}_p$ preserves the exploited mapping type semantics.

A.3 Semantics Preservation of Function $\mathcal{D}_o(t, m)$

The function $\mathcal{D}_o(t, m)$ was presented in Table 5.4 and it is used for the rewriting of a data triple pattern t based on a mapping m for its object part. Following the $\mathcal{D}_o$ function definition, in order to prove that the rewriting process preserves the exploited mapping type semantics, it suffices to consider the cases of triple patterns containing either a class or an individual in their object position.

To begin with, we prove that data triple pattern rewriting based on class mappings and function $\mathcal{D}_o$ is semantics preserving. To this end, let C_1 be a class, let $t = (sub, rdf{:}type, C_1)$ be a data triple pattern and let $\mathcal{J}$ be the set of variables appearing in t. For the different types of class mappings, we consider the following cases:

1. Given a named class c_1 and a mapping m of type $C_1 \equiv c_1$, stating that $C_1^{\mathcal{I}} = c_1^{\mathcal{I}}$, the triple pattern t is reformulated as follows:

$$\mathcal{D}_o(t, m) = (sub, rdf{:}type, c_1) = g$$

Furthermore, regarding the evaluation of the resulted graph pattern g over the RDF dataset DS (specified in the Definition A.1) applies that:

$$[[g]]_{DS} = [[(sub, rdf{:}type, c_1)]]_{DS}$$

Based on the semantics of the input triple pattern t, the mapping type m and the graph pattern g, we consider the following premises:

(a) $\forall\, solution\ \omega \in [[t]]_{DS}$: $\exists x$, such that $(x, rdf{:}type, C_1) \in DS$, and thus $x \in C_1^{\mathcal{I}}$. Taking into account that $C_1^{\mathcal{I}} = c_1^{\mathcal{I}}$ we conclude that $x \in c_1^{\mathcal{I}}$ and $(x, rdf{:}type, c_1) \in DS$. Therefore, $\omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

(b) $\forall\, solution\ \omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$: $\exists x$, such that $(x, rdf{:}type, c_1) \in DS$, and thus $x \in c_1^{\mathcal{I}}$. Taking into account that $C_1^{\mathcal{I}} = c_1^{\mathcal{I}}$ we conclude that $x \in C_1^{\mathcal{I}}$ and $(x, rdf{:}type, C_1) \in DS$. Therefore, $\omega \in [[t]]_{DS}$.

From the two premises above, we derive that $[[t]]_{DS} \equiv \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

2. Given two class expressions C_2, C_3 and a mapping m of type $C_1 \equiv C_2 \sqcap C_3$, stating that $C_1^{\mathcal{I}} = C_2^{\mathcal{I}} \cap C_3^{\mathcal{I}}$, the triple pattern t is reformulated as follows:

$$\mathcal{D}_o(t, m) = \mathcal{D}_o\big(t,\ C_1 \rightarrow C_2\big)\ \text{AND}\ \mathcal{D}_o\big(t,\ C_1 \rightarrow C_3\big) = g$$

Furthermore, regarding the evaluation of the resulted graph pattern g over the RDF dataset DS (specified in the Definition A.1) applies that:

$$\begin{aligned}[[g]]_{DS} &= [[\mathcal{D}_o\big(t,\ C_1 \rightarrow C_2\big)\ \text{AND}\ \mathcal{D}_o\big(t,\ C_1 \rightarrow C_3\big)]]_{DS} \\ &= [[\mathcal{D}_o\big(t,\ C_1 \rightarrow C_2\big)]]_{DS} \bowtie [[\mathcal{D}_o\big(t,\ C_1 \rightarrow C_3\big)]]_{DS}\end{aligned}$$

Abusively treating the class expressions C_2, C_3 as simple classes, and based on the semantics of the input triple pattern t, the mapping type m and the graph pattern g, we consider the following premises:

(a) $\forall\, solution\ \omega \in [[t]]_{DS}$: $\exists x$, such that $(x, rdf{:}type, C_1) \in DS$, and thus $x \in C_1^{\mathcal{I}}$. Taking into account that $C_1^{\mathcal{I}} = C_2^{\mathcal{I}} \cap C_3^{\mathcal{I}}$ we conclude that $(x, rdf{:}type, C_2) \in DS$ and $(x, rdf{:}type, C_3) \in DS$. To this end, $\omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

(b) $\forall\, solution\ \omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$: $\exists x$, so that both $(x, rdf{:}type, C_2), (x, rdf{:}type, C_3) \in DS$, and thus $x \in C_2^{\mathcal{I}} \cap C_3^{\mathcal{I}}$. Taking into account that $C_1^{\mathcal{I}} = C_2^{\mathcal{I}} \cap C_3^{\mathcal{I}}$ we conclude that $x \in C_1^{\mathcal{I}}$ and $(x, rdf{:}type, C_1) \in DS$. Therefore, $\omega \in [[t]]_{DS}$.

From the two premises above, we derive that $[[t]]_{DS} \equiv \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

3. Given two class expressions C_2, C_3 and a mapping m of type $C_1 \equiv C_2 \sqcup C_3$, stating that $C_1^{\mathcal{I}} = C_2^{\mathcal{I}} \cup C_3^{\mathcal{I}}$, the triple pattern t is reformulated as follows:

$$\mathcal{D}_o(t, m) = \mathcal{D}_o\big(t,\ C_1 \rightarrow C_2\big)\ \text{UNION}\ \mathcal{D}_o\big(t,\ C_1 \rightarrow C_3\big) = g$$

Furthermore, regarding the evaluation of the resulted graph pattern g over the RDF dataset DS (specified in the Definition A.1) applies that:

$$\begin{aligned}[[g]]_{DS} &= [[\mathcal{D}_o\big(t,\ C_1 \rightarrow C_2\big)\ \text{UNION}\ \mathcal{D}_o\big(t,\ C_1 \rightarrow C_3\big)]]_{DS} \\ &= [[\mathcal{D}_o\big(t,\ C_1 \rightarrow C_2\big)]]_{DS} \cup [[\mathcal{D}_o\big(t,\ C_1 \rightarrow C_3\big)]]_{DS}\end{aligned}$$

Abusively treating the class expressions C_2, C_3 as simple classes, and based on the semantics

of the input triple pattern t, the mapping type m and the graph pattern g, we consider the following premises:

(a) $\forall\, solution\ \omega \in [[t]]_{DS}$: $\exists x$, such that $(x, rdf{:}type, C_1) \in DS$, and thus $x \in C_1^{\mathcal{I}}$. Taking into account that $C_1^{\mathcal{I}} = C_2^{\mathcal{I}} \cup C_3^{\mathcal{I}}$ we conclude that $(x, rdf{:}type, C_2) \in DS$ or $(x, rdf{:}type, C_3) \in DS$. To this end, $\omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

(b) $\forall\, solution \omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$: $\exists x$, so that $(x, rdf{:}type, C_2) \in DS$ or $(x, rdf{:}type, C_3) \in DS$, and thus $x \in C_2^{\mathcal{I}} \cup C_3^{\mathcal{I}}$. Taking into account that $C_1^{\mathcal{I}} = C_2^{\mathcal{I}} \cup C_3^{\mathcal{I}}$ we conclude that $x \in C_1^{\mathcal{I}}$ and $(x, rdf{:}type, C_1) \in DS$. Therefore, $\omega \in [[t]]_{DS}$.

From the two premises above, we derive that $[[t]]_{DS} \equiv \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

4. Given two class expressions C_2, C_3 and a mapping m of type $C_1 \equiv C_2 - C_3$, stating that $C_1^{\mathcal{I}} = C_2^{\mathcal{I}} \setminus C_3^{\mathcal{I}}$, the triple pattern t is reformulated as follows:

$$\mathcal{D}_o(t, m) = \mathcal{D}_o\big(t,\ C_1 \to C_2\big) \text{ MINUS } \mathcal{D}_o\big(t,\ C_1 \to C_3\big) = g$$

Furthermore, regarding the evaluation of the resulted graph pattern g over the RDF dataset DS (specified in the Definition A.1) applies that:

$$\begin{aligned}[[g]]_{DS} &= [[\mathcal{D}_o\big(t,\ C_1 \to C_2\big) \text{ MINUS } \mathcal{D}_o\big(t,\ C_1 \to C_3\big)]]_{DS} \\ &= [[\mathcal{D}_o\big(t,\ C_1 \to C_2\big)]]_{DS} \setminus [[\mathcal{D}_o\big(t,\ C_1 \to C_3\big)]]_{DS}\end{aligned}$$

Abusively treating the class expressions C_2, C_3 as simple classes, and based on the semantics of the input triple pattern t, the mapping type m and the graph pattern g, we consider the following premises:

(a) $\forall\, solution\ \omega \in [[t]]_{DS}$: $\exists x$, such that $(x, rdf{:}type, C_1) \in DS$, and thus $x \in C_1^{\mathcal{I}}$. Taking into account that $C_1^{\mathcal{I}} = C_2^{\mathcal{I}} \setminus C_3^{\mathcal{I}}$ we conclude that $(x, rdf{:}type, C_2) \in DS$ and $(x, rdf{:}type, C_3) \notin DS$. To this end, $\omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

(b) $\forall\, solution\, \omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$: $\exists x$, such that $(x, rdf{:}type, C_2) \in DS$ and $(x, rdf{:}type, C_3) \notin DS$. Thus, $x \in C_2^{\mathcal{I}} \setminus C_3^{\mathcal{I}}$. Taking into account that $C_1^{\mathcal{I}} = C_2^{\mathcal{I}} \setminus C_3^{\mathcal{I}}$ we conclude that $x \in C_1^{\mathcal{I}}$ and $(x, rdf{:}type, C_1) \in DS$. Therefore, $\omega \in [[t]]_{DS}$.

From the two premises above, we derive that $[[t]]_{DS} \equiv \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

5. Given an object property expression R_1, a class expression C_2 and a mapping m of type $C_1 \equiv \exists R_1.C_2$, stating that $C_1^{\mathcal{I}} = \mathcal{L} = \{\alpha \in \Delta^{\mathcal{I}} \mid \exists b.\ (\alpha, b) \in R_1^{\mathcal{I}} \wedge b \in C_2^{\mathcal{I}}\}$, the triple pattern t is reformulated as follows:

$$\mathcal{D}_o(t, m) = \mathcal{D}_p\big((sub, R, ?v),\ R \to R_1 \lfloor C_2\big) = g$$

Furthermore, regarding the evaluation of the resulted graph pattern g over the RDF dataset DS (specified in the Definition A.1) applies that:

$$[[g]]_{DS} = [[\mathcal{D}_p\big((sub, R, ?v),\ R \to R_1 \restriction C_2\big)]]_{DS}$$

Abusively treating the expressions R_1, C_2 as simple property and class respectively, and based on the semantics of the input triple pattern t, the mapping type m and the graph pattern g, we consider the following premises:

(a) $\forall\, solution\ \omega \in [[t]]_{DS}$: $\exists x$, such that $(x, rdf{:}type, C_1) \in DS$, and thus $x \in C_1^{\mathcal{I}}$. Taking into account that $C_1^{\mathcal{I}} = \mathcal{L}$ we conclude that $\exists y$ such that $(x, R_1, y) \in DS$ and $(y, rdf{:}type, C_2) \in DS$. To this end, $\omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

(b) $\forall\, solution\ \omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$: $\exists x, \exists y$, so that $(x, R_1, y), (y, rdf{:}type, C_2) \in DS$, and thus $x \in \mathcal{L}$. Taking into account that $C_1^{\mathcal{I}} = \mathcal{L}$ we conclude that $x \in C_1^{\mathcal{I}}$ and $(x, rdf{:}type, C_1) \in DS$. Therefore, $\omega \in [[t]]_{DS}$.

From the two premises above, we derive that $[[t]]_{DS} \equiv \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

6. Given a datatype property expression U_1, a data range D_1 and a mapping m of type $C_1 \equiv \exists U_1.D_1$, stating that $C_1^{\mathcal{I}} = \mathcal{L} = \{\alpha \in \Delta^{\mathcal{I}} \mid \exists b.\ (\alpha, b) \in U_1^{\mathcal{I}} \wedge b \in D_1^{\mathcal{D}}\}$, the triple pattern t is reformulated as follows:

$$\mathcal{D}_o(t, m) = \mathcal{D}_p\big((sub, U, ?v),\ U \to U_1 \restriction D_1\big) = g$$

Furthermore, regarding the evaluation of the resulted graph pattern g over the RDF dataset DS (specified in the Definition A.1) applies that:

$$[[g]]_{DS} = [[\mathcal{D}_p\big((sub, U, ?v),\ U \to U_1 \restriction D_1\big)]]_{DS}$$

Abusively treating the datatype property expression U_1 as a simple datatype property, and based on the semantics of the input triple pattern t, the mapping type m and the graph pattern g, we consider the following premises:

(a) $\forall\, solution\ \omega \in [[t]]_{DS}$: $\exists x$, such that $(x, rdf{:}type, C_1) \in DS$, and thus $x \in C_1^{\mathcal{I}}$. Taking into account that $C_1^{\mathcal{I}} = \mathcal{L}$ we conclude that $\exists y \in D_1^{\mathcal{D}}$ such that $(x, U_1, y) \in DS$. To this end, $\omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

(b) $\forall\, solution\ \omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$: $\exists x$ and $\exists y \in D_1^{\mathcal{D}}$, such that $(x, U_1, y) \in DS$, and thus $x \in \mathcal{L}$. Taking into account that $C_1^{\mathcal{I}} = \mathcal{L}$ we conclude that $x \in C_1^{\mathcal{I}}$ and $(x, rdf{:}type, C_1) \in DS$. Therefore, $\omega \in [[t]]_{DS}$.

From the two premises above, we derive that $[[t]]_{DS} \equiv \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

7. Given an object property expression R_1 and a mapping m of type $C_1 \equiv \exists R_1.P_1$, stating that $C_1^{\mathcal{I}} = \mathcal{L} = \{\alpha \in \Delta^{\mathcal{I}} \mid \exists b.\ (\alpha, b) \in R_1^{\mathcal{I}} \wedge b \in P_1^{\mathcal{D}}\}$, the triple pattern t is reformulated as follows:

$$\mathcal{D}_o(t, m) = \mathcal{D}_p\big((sub, R, ?v),\ R \rightarrow R_1\big) \text{ FILTER } \mathcal{T}_u\big(P_1, ?v\big) = g$$

Furthermore, regarding the evaluation of the resulted graph pattern g over the RDF dataset DS (specified in the Definition A.1) applies that:

$$\begin{aligned}[[g]]_{DS} &= [[\mathcal{D}_p\big((sub, R, ?v),\ R \rightarrow R_1\big) \text{ FILTER } \mathcal{T}_u\big(P_1, ?v\big)]]_{DS} \\ &= \{\omega \in [[\mathcal{D}_p\big((sub, R, ?v),\ R \rightarrow R_1\big)]]_{DS} \mid \omega \models \mathcal{T}_u\big(P_1, ?v\big)\}\end{aligned}$$

Abusively treating the object property expression R_1 as a simple object property, and based on the semantics of the input triple pattern t, the mapping type m and the graph pattern g, we consider the following premises:

(a) $\forall\, solution\ \omega \in [[t]]_{DS}$: $\exists x$, such that $(x, rdf{:}type, C_1) \in DS$, and thus $x \in C_1^{\mathcal{I}}$. Taking into account that $C_1^{\mathcal{I}} = \mathcal{L}$ we conclude that $\exists y \in P_1^{\mathcal{D}}$ such that $(x, R_1, y) \in DS$. To this end, $\omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

(b) $\forall\, solution\ \omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$: $\exists x$ and $\exists y \in P_1^{\mathcal{D}}$, such that $(x, R_1, y) \in DS$, and thus $x \in \mathcal{L}$. Taking into account that $C_1^{\mathcal{I}} = \mathcal{L}$ we conclude that $x \in C_1^{\mathcal{I}}$ and $(x, rdf{:}type, C_1) \in DS$. Therefore, $\omega \in [[t]]_{DS}$.

From the two premises above, we derive that $[[t]]_{DS} \equiv \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

8. Given a datatype property expression U_1 and a mapping m of type $C_1 \equiv \exists U_1.P_1$, stating that $C_1^{\mathcal{I}} = \mathcal{L} = \{\alpha \in \Delta^{\mathcal{I}} \mid \exists b.\ (\alpha, b) \in U_1^{\mathcal{I}} \wedge b \in P_1^{\mathcal{D}}\}$, the triple pattern t is reformulated as follows:

$$\mathcal{D}_o(t, m) = \mathcal{D}_p\big((sub, U, ?v),\ U \rightarrow U_1\big) \text{ FILTER } \mathcal{T}_u\big(P_1, ?v\big) = g$$

Furthermore, regarding the evaluation of the resulted graph pattern g over the RDF dataset DS (specified in the Definition A.1) applies that:

$$\begin{aligned}[[g]]_{DS} &= [[\mathcal{D}_p\big((sub, U, ?v),\ U \rightarrow U_1\big) \text{ FILTER } \mathcal{T}_u\big(P_1, ?v\big)]]_{DS} \\ &= \{\omega \in [[\mathcal{D}_p\big((sub, U, ?v),\ U \rightarrow U_1\big)]]_{DS} \mid \omega \models \mathcal{T}_u\big(P_1, ?v\big)\}\end{aligned}$$

Abusively treating the datatype property expression U_1 as a simple datatype property, and based on the semantics of the input triple pattern t, the mapping type m and the graph pattern g, we consider the following premises:

(a) $\forall\, solution\ \omega \in [[t]]_{DS}$: $\exists x$, such that $(x, rdf{:}type, C_1) \in DS$, and thus $x \in C_1^{\mathcal{I}}$. Taking into account that $C_1^{\mathcal{I}} = \mathcal{L}$ we conclude that $\exists y \in P_1^{\mathcal{D}}$ such that $(x, U_1, y) \in DS$. To this end, $\omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

(b) $\forall\, solution\ \omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$: $\exists x$ and $\exists y \in P_1^{\mathcal{D}}$, such that $(x, U_1, y) \in DS$, and thus $x \in \mathcal{L}$. Taking into account that $C_1^{\mathcal{I}} = \mathcal{L}$ we conclude that $x \in C_1^{\mathcal{I}}$ and $(x, rdf{:}type, C_1) \in DS$. Therefore, $\omega \in [[t]]_{DS}$.

From the two premises above, we derive that $[[t]]_{DS} \equiv \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

9. Given two object property expressions R_1, R_2 and a mapping m of type $C_1 \equiv \exists(R_1, R_2).P_1$, stating that $C_1^{\mathcal{I}} = \mathcal{L} = \{\alpha \in \Delta^{\mathcal{I}} \mid \exists b_1, b_2.\ (\alpha, b_1) \in R_1^{\mathcal{I}} \wedge (\alpha, b_2) \in R_2^{\mathcal{I}} \wedge (b_1, b_2) \in P_1^{\mathcal{D}}\}$, the triple pattern t is reformulated as follows:

$$\begin{aligned}\mathcal{D}_o(t, m) = \mathcal{D}_p\big((sub, R, ?v_1),\ R \to R_1\big) \text{ AND } \mathcal{D}_p\big((sub, R, ?v_2),\ R \to R_2\big)\\ \text{FILTER } \mathcal{T}_b\big(P_1, ?v_1, ?v_2\big) = g\end{aligned}$$

Furthermore, regarding the evaluation of the resulted graph pattern g over the RDF dataset DS (specified in the Definition A.1) applies that:

$$\begin{aligned}[[g]]_{DS} &= [[\mathcal{D}_p\big((sub, R, ?v_1),\ R \to R_1\big) \text{ AND } \mathcal{D}_p\big((sub, R, ?v_2),\ R \to R_2\big)\\ &\quad \text{FILTER } \mathcal{T}_b\big(P_1, ?v_1, ?v_2\big)]]_{DS}\\ &= \{\omega \in [[\mathcal{D}_p\big((sub, R, ?v_1),\ R \to R_1\big)]]_{DS} \bowtie [[\mathcal{D}_p\big((sub, R, ?v_2),\\ &\quad R \to R_2\big)]]_{DS} \mid \omega \models \mathcal{T}_b\big(P_1, ?v_1, ?v_2\big)\}\end{aligned}$$

Abusively treating the object property expressions R_1, R_2 as simple object properties, and based on the semantics of the input triple pattern t, the mapping type m and the graph pattern g, we consider the following premises:

(a) $\forall\, solution\, \omega \in [[t]]_{DS}$: $\exists x$, such that $(x, rdf{:}type, C_1) \in DS$, and thus $x \in C_1^{\mathcal{I}}$. Taking into account that $C_1^{\mathcal{I}} = \mathcal{L}$ we conclude that $\exists y_1, \exists y_2$ so that $(x, R_1, y_1), (x, R_2, y_2) \in DS$ and $(y_1, y_2) \in P_1^{\mathcal{D}}$. To this end, $\omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

(b) $\forall\, solution\, \omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$: $\exists x, \exists y_1, \exists y_2$, such that $(x, R_1, y_1) \in DS$, $(x, R_2, y_2) \in DS$ and $(y_1, y_2) \in P_1^{\mathcal{D}}$. Thus, $x \in \mathcal{L}$. Taking into account that $C_1^{\mathcal{I}} = \mathcal{L}$ we conclude that $x \in C_1^{\mathcal{I}}$ and $(x, rdf{:}type, C_1) \in DS$. Therefore, $\omega \in [[t]]_{DS}$.

From the two premises above, we derive that $[[t]]_{DS} \equiv \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

10. Given two datatype property expressions U_1, U_2, as well as a mapping m of type $C_1 \equiv \exists(U_1, U_2).P_1$, stating that $C_1^{\mathcal{I}} = \mathcal{L} = \{\alpha \in \Delta^{\mathcal{I}} \mid \exists b_1, b_2.\ (\alpha, b_1) \in U_1^{\mathcal{I}} \wedge (\alpha, b_2) \in U_2^{\mathcal{I}} \wedge (b_1, b_2) \in P_1^{\mathcal{D}}\}$, the triple pattern t is reformulated as follows:

$$\begin{aligned}\mathcal{D}_o(t, m) = \mathcal{D}_p\big((sub, U, ?v_1),\ U \to U_1\big) \text{ AND } \mathcal{D}_p\big((sub, U, ?v_2),\ U \to U_2\big)\\ \text{FILTER } \mathcal{T}_b\big(P_1, ?v_1, ?v_2\big) = g\end{aligned}$$

Furthermore, regarding the evaluation of the resulted graph pattern g over the RDF dataset DS (specified in the Definition A.1) applies that:

$$\begin{aligned}[[g]]_{DS} &= [[\mathcal{D}_p\big((sub, U, ?v_1),\ U \rightarrow U_1\big) \text{ AND } \mathcal{D}_p\big((sub, U, ?v_2),\ U \rightarrow U_2\big) \\ &\quad \text{FILTER } \mathcal{T}_b\big(P_1, ?v_1, ?v_2\big)]]_{DS} \\ &= \{\omega \in [[\mathcal{D}_p\big((sub, U, ?v_1),\ U \rightarrow U_1\big)]]_{DS} \bowtie [[\mathcal{D}_p\big((sub, U, ?v_2), \\ &\quad U \rightarrow U_2\big)]]_{DS} \mid \omega \models \mathcal{T}_b\big(P_1, ?v_1, ?v_2\big)\}\end{aligned}$$

Abusively treating the datatype property expressions U_1, U_2 as simple datatype properties, and based on the semantics of the input triple pattern t, the mapping type m and the graph pattern g, we consider the following premises:

(a) $\forall\, solution\, \omega \in [[t]]_{DS}$: $\exists x$, such that $(x, rdf{:}type, C_1) \in DS$, and thus $x \in C_1^{\mathcal{I}}$. Taking into account that $C_1^{\mathcal{I}} = \mathcal{L}$ we conclude that $\exists y_1, \exists y_2$ so that $(x, U_1, y_1), (x, U_2, y_2) \in DS$ and $(y_1, y_2) \in P_1^{\mathcal{D}}$. To this end, $\omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

(b) $\forall\, solution\, \omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$: $\exists x, \exists y_1, \exists y_2$, such that $(x, U_1, y_1) \in DS$, $(x, U_2, y_2) \in DS$ and $(y_1, y_2) \in P_1^{\mathcal{D}}$. Thus, $x \in \mathcal{L}$. Taking into account that $C_1^{\mathcal{I}} = \mathcal{L}$ we conclude that $x \in C_1^{\mathcal{I}}$ and $(x, rdf{:}type, C_1) \in DS$. Therefore, $\omega \in [[t]]_{DS}$.

From the two premises above, we derive that $[[t]]_{DS} \equiv \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

11. Given an object property expression R_1 and a mapping m of type $C_1 \equiv \geqq\!\!\!\!\!\!= n\ R_1$, stating that $C_1^{\mathcal{I}} = \mathcal{L} = \{\alpha \in \Delta^{\mathcal{I}} \mid |\{b \in \Delta^{\mathcal{I}} \mid (\alpha, b) \in R_1^{\mathcal{I}}\}| \ \geqq\!\!\!\!\!\!= n\}$, the triple pattern t is reformulated as follows:

$$\begin{aligned}\mathcal{D}_o(t, m) = \text{SELECT DISTINCT } (sub) \text{ WHERE } &\big(\mathcal{D}_p\big((sub, R, ?v),\ R \rightarrow R_1\big)\big) \\ \text{GROUP BY } (sub) \text{ HAVING } &\big(\text{COUNT (DISTINCT } ?v) \ \geqq\!\!\!\!\!\!= n\big) = g\end{aligned}$$

Furthermore, regarding the evaluation of the resulted graph pattern g over the RDF dataset DS (specified in the Definition A.1) applies that:

$$\begin{aligned}[[g]]_{DS} = [[\text{SELECT DISTINCT } (sub) \text{ WHERE } &\big(\mathcal{D}_p\big((sub, R, ?v),\ R \rightarrow R_1\big)\big) \\ \text{GROUP BY } (sub) \text{ HAVING } &\big(\text{COUNT (DISTINCT } ?v) \ \geqq\!\!\!\!\!\!= n\big)]]_{DS}\end{aligned}$$

Abusively treating the object property expression R_1 as a simple object property, and based on the semantics of the input triple pattern t, the mapping type m and the graph pattern g, we consider the following premises:

(a) $\forall\, solution\ \omega \in [[t]]_{DS}$: $\exists x$, such that $(x, rdf{:}type, C_1) \in DS$, and thus $x \in C_1^{\mathcal{I}}$. Taking into account that $C_1^{\mathcal{I}} = \mathcal{L}$ we conclude that there exist $\geqq\!\!\!\!\!\!= n$ values of y so

that for each different value of y applies that $(x, R_1, y) \in DS$. To this end, $\omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

(b) $\forall\, solution\ \omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$: $\exists x$, and also exist $\overset{\geq}{\underset{\leq}{=}}\ n$ values of y so that for each different value of y applies that $(x, R_1, y) \in DS$. Thus, $x \in \mathcal{L}$. Taking into account that $C_1^{\mathcal{I}} = \mathcal{L}$ we conclude that $x \in C_1^{\mathcal{I}}$ and $(x, rdf{:}type, C_1) \in DS$. Therefore, $\omega \in [[t]]_{DS}$.

From the two premises above, we derive that $[[t]]_{DS} \equiv \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

12. Given an datatype property expression U_1 and a mapping m of type $C_1 \equiv \overset{\geq}{\underset{\leq}{=}}\ n\ U_1$, stating that $C_1^{\mathcal{I}} = \mathcal{L} = \{\alpha \in \Delta^{\mathcal{I}} \mid |\{b \in \Delta^{\mathcal{I}} \mid (\alpha, b) \in U_1^{\mathcal{I}}\}| \overset{\geq}{\underset{\leq}{=}}\ n\}$, the triple pattern t is reformulated as follows:

$$\mathcal{D}_o(t, m) = \text{SELECT DISTINCT } (sub) \text{ WHERE } \big(\mathcal{D}_p\big((sub, U, ?v),\ U \rightarrow U_1\big)\big)$$
$$\text{GROUP BY } (sub) \text{ HAVING } \big(\text{COUNT (DISTINCT } ?v) \overset{\geq}{\underset{\leq}{=}}\ n\big) = g$$

Furthermore, regarding the evaluation of the resulted graph pattern g over the RDF dataset DS (specified in the Definition A.1) applies that:

$$[[g]]_{DS} = [[\text{SELECT DISTINCT } (sub) \text{ WHERE } \big(\mathcal{D}_p\big((sub, U, ?v),\ U \rightarrow U_1\big)\big)$$
$$\text{GROUP BY } (sub) \text{ HAVING } \big(\text{COUNT (DISTINCT } ?v) \overset{\geq}{\underset{\leq}{=}}\ n\big)]]_{DS}$$

Abusively treating the datatype property expression U_1 as a simple datatype property, and based on the semantics of the input triple pattern t, the mapping type m and the graph pattern g, we consider the following premises:

(a) $\forall\, solution\ \omega \in [[t]]_{DS}$: $\exists x$, such that $(x, rdf{:}type, C_1) \in DS$, and thus $x \in C_1^{\mathcal{I}}$. Taking into account that $C_1^{\mathcal{I}} = \mathcal{L}$ we conclude that there exist $\overset{\geq}{\underset{\leq}{=}}\ n$ values of y so that for each different value of y applies that $(x, U_1, y) \in DS$. To this end, $\omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

(b) $\forall\, solution\ \omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$: $\exists x$, and also exist $\overset{\geq}{\underset{\leq}{=}}\ n$ values of y so that for each different value of y applies that $(x, U_1, y) \in DS$. Thus, $x \in \mathcal{L}$. Taking into account that $C_1^{\mathcal{I}} = \mathcal{L}$ we conclude that $x \in C_1^{\mathcal{I}}$ and $(x, rdf{:}type, C_1) \in DS$. Therefore, $\omega \in [[t]]_{DS}$.

From the two premises above, we derive that $[[t]]_{DS} \equiv \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

13. Given an object property expression R_1, a class expression C_2 and a mapping m of type $C_1 \equiv \overset{\geq}{\underset{\leq}{=}}\ n\ R_1.C_2$, stating that $C_1^{\mathcal{I}} = \mathcal{L} = \{\alpha \in \Delta^{\mathcal{I}} \mid |\{b \in \Delta^{\mathcal{I}} \mid (\alpha, b) \in R_1^{\mathcal{I}} \wedge b \in C_2^{\mathcal{I}}\}| \geq$

$n\}$, the triple pattern t is reformulated as follows:

$$\mathcal{D}_o(t,m) = \text{SELECT DISTINCT}(sub)\ \text{WHERE}\big(\mathcal{D}_p\big((sub, R, ?v), R \rightarrow R_1 \lfloor C_2\big)\big)$$
$$\text{GROUP BY}\ (sub)\ \text{HAVING}\ \big(\text{COUNT}\ (\text{DISTINCT}\ ?v)\ \gtreqless n\big) = g$$

Furthermore, regarding the evaluation of the resulted graph pattern g over the RDF dataset DS (specified in the Definition A.1) applies that:

$$[[g]]_{DS} = [[\text{SELECT DISTINCT}\ (sub)\ \text{WHERE}\ \big(\mathcal{D}_p\big((sub, R, ?v), R \rightarrow R_1 \lfloor C_2\big)\big)$$
$$\text{GROUP BY}\ (sub)\ \text{HAVING}\ \big(\text{COUNT}\ (\text{DISTINCT}\ ?v)\ \gtreqless n\big)]]_{DS}$$

Abusively treating the expressions R_1, C_2 as simple property and class respectively, and based on the semantics of the input triple pattern t, the mapping type m and the graph pattern g, we consider the following premises:

(a) $\forall\, solution\ \omega \in [[t]]_{DS}$: $\exists x$, such that $(x, rdf{:}type, C_1) \in DS$, and thus $x \in C_1^{\mathcal{I}}$. Taking into account that $C_1^{\mathcal{I}} = \mathcal{L}$ we conclude that there exist $\gtreqless n$ values of y so that for each different value of y applies that $(x, R_1, y) \in DS$ and $(y, rdf{:}type, C_2) \in DS$. To this end, $\omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

(b) $\forall\, solution\ \omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$: $\exists x$, and also exist $\gtreqless n$ values of y so that for each different value of y applies that both $(x, R_1, y) \in DS$ and $(y, rdf{:}type, C_2) \in DS$. Thus, $x \in \mathcal{L}$. Taking into account that $C_1^{\mathcal{I}} = \mathcal{L}$ we conclude that $x \in C_1^{\mathcal{I}}$ and $(x, rdf{:}type, C_1) \in DS$. Therefore, $\omega \in [[t]]_{DS}$.

From the two premises above, we derive that $[[t]]_{DS} \equiv \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

14. Given a datatype property expression U_1, a data range D_1 and a mapping m of type $C_1 \equiv \gtreqless n\, U_1.D_1$, stating that $C_1^{\mathcal{I}} = \mathcal{L} = \{\alpha \in \Delta^{\mathcal{I}} \mid |\{b \in \Delta^{\mathcal{D}} \mid (\alpha, b) \in U_1^{\mathcal{I}} \wedge b \in D_1^{\mathcal{D}}\}| \geq n\}$, the triple pattern t is reformulated as follows:

$$\mathcal{D}_o(t,m) = \text{SELECT DISTINCT}(sub)\ \text{WHERE}\big(\mathcal{D}_p\big((sub, U, ?v), U \rightarrow U_1 \lfloor D_1\big)\big)$$
$$\text{GROUP BY}\ (sub)\ \text{HAVING}\ \big(\text{COUNT}\ (\text{DISTINCT}\ ?v)\ \gtreqless n\big) = g$$

Furthermore, regarding the evaluation of the resulted graph pattern g over the RDF dataset DS (specified in the Definition A.1) applies that:

$$[[g]]_{DS} = [[\text{SELECT DISTINCT}\ (sub)\ \text{WHERE}\ \big(\mathcal{D}_p\big((sub, U, ?v), U \rightarrow U_1 \lfloor D_1\big)\big)$$
$$\text{GROUP BY}\ (sub)\ \text{HAVING}\ \big(\text{COUNT}\ (\text{DISTINCT}\ ?v)\ \gtreqless n\big)]]_{DS}$$

Abusively treating the datatype property expression U_1 as a simple datatype property, and based on the semantics of the input triple pattern t, the mapping type m and the graph pattern

g, we consider the following premises:

(a) $\forall\, solution\ \omega \in [[t]]_{DS}$: $\exists x$, such that $(x, rdf{:}type, C_1) \in DS$, and thus $x \in C_1^{\mathcal{I}}$. Taking into account that $C_1^{\mathcal{I}} = \mathcal{L}$ we conclude that there exist $\overset{\geq}{\leq}\, n$ values of y so that for each different value of y applies that $(x, U_1, y) \in DS$ and $y \in D_1^{\mathcal{D}}$. To this end, $\omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

(b) $\forall\, solution\ \omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$: $\exists x$, and also exist $\overset{\geq}{\leq}\, n$ values of y so that for each different value of y applies that $(x, U_1, y) \in DS$ and $y \in D_1^{\mathcal{D}}$. Thus, $x \in \mathcal{L}$. Taking into account that $C_1^{\mathcal{I}} = \mathcal{L}$ we conclude that $x \in C_1^{\mathcal{I}}$ and $(x, rdf{:}type, C_1) \in DS$. Therefore, $\omega \in [[t]]_{DS}$.

From the two premises above, we derive that $[[t]]_{DS} \equiv \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

15. Given the individuals $o_1, \ldots, o_n$ and a mapping m of type $C_1 \equiv \{o_1, \ldots, o_n\}$, stating that $C_1^{\mathcal{I}} = \{o_1, \ldots, o_n\}^{\mathcal{I}}$, the triple pattern t is reformulated as follows:

$$\mathcal{D}_o(t, m) = \text{VALUES } (sub)\ (o_1 \ldots o_n) = g$$

Furthermore, regarding the evaluation of the resulted graph pattern g over the RDF dataset DS (specified in the Definition A.1) applies that:

$$\begin{aligned}[[g]]_{DS} &= [[\text{VALUES } (sub)\ (o_1 \ldots o_n)]]_{DS} \\ &= \{\omega \mid dom(\omega) = sub,\ \omega(sub) = \{o_1, \ldots, o_n\}\}\end{aligned}$$

Based on the semantics of the input triple pattern t, the mapping type m and the graph pattern g, we consider the following premises:

(a) $\forall\, solution\ \omega \in [[t]]_{DS}$: $\exists x$, such that $(x, rdf{:}type, C_1) \in DS$, and thus $x \in C_1^{\mathcal{I}}$. Taking into account that $C_1^{\mathcal{I}} = \{o_1, \ldots, o_n\}^{\mathcal{I}}$ we conclude that $x \in \{o_1, \ldots, o_n\}^{\mathcal{I}}$. Therefore, $\omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

(b) $\forall\, solution\ \omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$: $\exists x$, such that $dom(\omega) = x$ and $\omega(x) \in \{o_1, \ldots, o_n\}^{\mathcal{I}}$. Taking into account that $C_1^{\mathcal{I}} = \{o_1, \ldots, o_n\}^{\mathcal{I}}$, we derive that $x \in C_1^{\mathcal{I}}$, as well as the fact that $(x, rdf{:}type, C_1) \in DS$. Therefore, $\omega \in [[t]]_{DS}$.

From the two premises above, we derive that $[[t]]_{DS} \equiv \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.

Similarly, we prove that data triple pattern rewriting using the function $\mathcal{D}_o$ along with an individual mapping is semantics preserving. To this end, let o_1, o_2 be individuals, let $t = (sub, pred, o_1)$ be a data triple pattern and let $\mathcal{J}$ be the set of variables appearing in t. Given a mapping m of type $o_1 \equiv o_2$, stating that $o_1^{\mathcal{I}} = o_2^{\mathcal{I}}$, the triple pattern t is reformulated as follows:

$$\mathcal{D}_o(t, m) = (sub, pred, o_2) = g$$

Furthermore, regarding the evaluation of the resulted graph pattern g over the RDF dataset DS (specified in the Definition A.1) applies that:

$$[[g]]_{DS} = [[(sub, pred, o_2)]]_{DS}$$

Based on the semantics of the input triple pattern t, the mapping type m and the graph pattern g, we consider the following premises:

1. $\forall\, solution\, \omega \in [[t]]_{DS}\colon\ \exists x, \exists y$, such that $(x, y, o_1) \in DS$. Taking into account that $o_1^{\mathcal{I}} = o_2^{\mathcal{I}}$ we conclude that $(x, y, o_2) \in DS$, and therefore, $\omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$.
2. $\forall\, solution\, \omega \in \pi_{\mathcal{J}}\big([[g]]_{DS}\big)\colon\ \exists x, \exists y$, such that $(x, y, o_2) \in DS$. Taking into account that $o_1^{\mathcal{I}} = o_2^{\mathcal{I}}$ we conclude that $(x, y, o_1) \in DS$, and therefore, $\omega \in [[t]]_{DS}$.

From the two premises above, we derive that $[[t]]_{DS} \equiv \pi_{\mathcal{J}}\big([[g]]_{DS}\big)$. This concludes the proof that the function $\mathcal{D}_o$ preserves the exploited mapping type semantics.

Appendix B

Mapping Language: Schema

This appendix provides the XML Schema of the language which is used for the mapping representation in the context of the SPARQL–RW Framework. The schema is based on the grammar presented in Section 4.6, defining the core constructs of the model in XML format. It introduces language constraints and provides control over the type of data that can be assigned to the various elements and attributes. Furthermore, the use of XML and XML Schema in describing the SPARQL–RW mapping language: (*a*) provides exceptional *validation* capabilities, (*b*) supports easy mapping *serialization and deserialization*, and (*c*) enables *interoperability* with external systems and applications.

```
<xs:schema version="1.0"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:tns="http://www.music.tuc.gr/sparqlrw"
           targetNamespace="http://www.music.tuc.gr/sparqlrw"
           elementFormDefault="qualified">

   <!-- SPARQL-RW Mapping Model -->
   <xs:element name="model">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="tns:formalism"/>
            <xs:element name="global">
               <xs:complexType>
                  <xs:sequence>
                     <xs:element ref="tns:ontology"/>
                  </xs:sequence>
               </xs:complexType>
            </xs:element>
            <xs:element name="locals">
               <xs:complexType>
                  <xs:sequence>
                     <xs:element ref="tns:ontology"
```

```
                              maxOccurs="unbounded"/>
               </xs:sequence>
            </xs:complexType>
         </xs:element>
         <xs:element name="mappings">
            <xs:complexType>
               <xs:sequence maxOccurs="unbounded">
                  <xs:choice>
                     <xs:element ref="tns:cmapping"/>
                     <xs:element ref="tns:opmapping"/>
                     <xs:element ref="tns:dpmapping"/>
                     <xs:element ref="tns:imapping"/>
                  </xs:choice>
               </xs:sequence>
            </xs:complexType>
         </xs:element>
      </xs:sequence>
   </xs:complexType>
</xs:element>

<!-- Ontology -->
<xs:element name="ontology">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="name" type="xs:string"/>
         <xs:element name="description" type="xs:string"/>
      </xs:sequence>
      <xs:attribute ref="tns:uri" use="required"/>
   </xs:complexType>
</xs:element>

<!-- Class Mapping -->
<xs:element name="cmapping">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="expr1">
            <xs:complexType>
               <xs:sequence>
                  <xs:element ref="tns:class"/>
               </xs:sequence>
            </xs:complexType>
         </xs:element>
         <xs:element name="expr2">
            <xs:complexType>
               <xs:sequence>
                  <xs:element ref="tns:class"/>
               </xs:sequence>
            </xs:complexType>
         </xs:element>
         <xs:element ref="tns:relation"/>
```

```
      </xs:sequence>
      <xs:attribute ref="tns:uri" use="required"/>
   </xs:complexType>
</xs:element>

<!-- Object Property Mapping -->
<xs:element name="opmapping">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="expr1">
            <xs:complexType>
               <xs:sequence>
                  <xs:element ref="tns:oproperty"/>
               </xs:sequence>
            </xs:complexType>
         </xs:element>
         <xs:element name="expr2">
            <xs:complexType>
               <xs:sequence>
                  <xs:element ref="tns:oproperty"/>
               </xs:sequence>
            </xs:complexType>
         </xs:element>
         <xs:element ref="tns:relation"/>
      </xs:sequence>
      <xs:attribute ref="tns:uri" use="required"/>
   </xs:complexType>
</xs:element>

<!-- Datatype Property Mapping -->
<xs:element name="dpmapping">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="expr1">
            <xs:complexType>
               <xs:sequence>
                  <xs:element ref="tns:dproperty"/>
               </xs:sequence>
            </xs:complexType>
         </xs:element>
         <xs:element name="expr2">
            <xs:complexType>
               <xs:sequence>
                  <xs:element ref="tns:dproperty"/>
               </xs:sequence>
            </xs:complexType>
         </xs:element>
         <xs:element ref="tns:relation"/>
      </xs:sequence>
```

```
      <xs:attribute ref="tns:uri" use="required"/>
   </xs:complexType>
</xs:element>

<!-- Individual Mapping -->
<xs:element name="imapping">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="expr1">
            <xs:complexType>
               <xs:sequence>
                  <xs:element ref="tns:individual"/>
               </xs:sequence>
            </xs:complexType>
         </xs:element>
         <xs:element name="expr2">
            <xs:complexType>
               <xs:sequence>
                  <xs:element ref="tns:individual"/>
               </xs:sequence>
            </xs:complexType>
         </xs:element>
         <xs:element ref="tns:relation"/>
      </xs:sequence>
      <xs:attribute ref="tns:uri" use="required"/>
   </xs:complexType>
</xs:element>

<!-- Class -->
<xs:element name="class">
   <xs:complexType>
      <xs:choice>
         <xs:element ref="tns:resource"/>
         <xs:element name="union">
            <xs:complexType>
               <xs:sequence>
                  <xs:element ref="tns:class" minOccurs="2"
                              maxOccurs="unbounded"/>
               </xs:sequence>
            </xs:complexType>
         </xs:element>
         <xs:element name="intersection">
            <xs:complexType>
               <xs:sequence>
                  <xs:element ref="tns:class" minOccurs="2"
                              maxOccurs="unbounded"/>
               </xs:sequence>
            </xs:complexType>
         </xs:element>
         <xs:element name="difference">
```

```
            <xs:complexType>
               <xs:sequence>
                  <xs:element ref="tns:class" minOccurs="2"
                              maxOccurs="unbounded"/>
               </xs:sequence>
            </xs:complexType>
         </xs:element>
         <xs:element name="enumeration">
            <xs:complexType>
               <xs:sequence>
                  <xs:element ref="tns:individual"
                              maxOccurs="unbounded"/>
               </xs:sequence>
            </xs:complexType>
         </xs:element>
         <xs:element name="existquant">
            <xs:complexType>
               <xs:choice>
                  <xs:element name="onOProperty">
                     <xs:complexType>
                        <xs:sequence>
                           <xs:element ref="tns:oproperty"/>
                           <xs:element name="quantifier">
                              <xs:complexType>
                                 <xs:sequence>
                                    <xs:element ref="tns:class"/>
                                 </xs:sequence>
                              </xs:complexType>
                           </xs:element>
                        </xs:sequence>
                     </xs:complexType>
                  </xs:element>
                  <xs:element name="onDProperty">
                     <xs:complexType>
                        <xs:sequence>
                           <xs:element ref="tns:dproperty"/>
                           <xs:element name="quantifier">
                              <xs:complexType>
                                 <xs:sequence>
                                    <xs:element ref="tns:datatype"/>
                                 </xs:sequence>
                              </xs:complexType>
                           </xs:element>
                        </xs:sequence>
                     </xs:complexType>
                  </xs:element>
               </xs:choice>
            </xs:complexType>
         </xs:element>
```

```
<xs:element name="existpred">
   <xs:complexType>
      <xs:choice>
         <xs:element name="onOProperty">
            <xs:complexType>
               <xs:sequence>
                  <xs:element ref="tns:oproperty"/>
                  <xs:element name="predicate"
                              type="tns:UPredicateType"/>
               </xs:sequence>
            </xs:complexType>
         </xs:element>
         <xs:element name="onDProperty">
            <xs:complexType>
               <xs:sequence>
                  <xs:element ref="tns:dproperty"/>
                  <xs:element name="predicate"
                              type="tns:UPredicateType"/>
               </xs:sequence>
            </xs:complexType>
         </xs:element>
         <xs:element name="onOProperties">
            <xs:complexType>
               <xs:sequence>
                  <xs:element ref="tns:oproperty"
                              minOccurs="2"
                              maxOccurs="2"/>
                  <xs:element name="predicate"
                              type="tns:BPredicateType"/>
               </xs:sequence>
            </xs:complexType>
         </xs:element>
         <xs:element name="onDProperties">
            <xs:complexType>
               <xs:sequence>
                  <xs:element ref="tns:dproperty"
                              minOccurs="2"
                              maxOccurs="2"/>
                  <xs:element name="predicate"
                              type="tns:BPredicateType"/>
               </xs:sequence>
            </xs:complexType>
         </xs:element>
      </xs:choice>
   </xs:complexType>
</xs:element>
<xs:element name="cardinality">
   <xs:complexType>
      <xs:choice>
```

```
                  <xs:element name="onOProperty">
                     <xs:complexType>
                        <xs:sequence>
                           <xs:element ref="tns:oproperty"/>
                           <xs:element name="predicate"
                                       type="tns:UPredicateType"/>
                           <xs:element name="quantifier"
                                       minOccurs="0">
                              <xs:complexType>
                                 <xs:sequence>
                                    <xs:element ref="tns:class"/>
                                 </xs:sequence>
                              </xs:complexType>
                           </xs:element>
                        </xs:sequence>
                     </xs:complexType>
                  </xs:element>
                  <xs:element name="onDProperty">
                     <xs:complexType>
                        <xs:sequence>
                           <xs:element ref="tns:dproperty"/>
                           <xs:element name="predicate"
                                       type="tns:UPredicateType"/>
                           <xs:element name="quantifier"
                                       minOccurs="0">
                              <xs:complexType>
                                 <xs:sequence>
                                    <xs:element ref="tns:datatype"/>
                                 </xs:sequence>
                              </xs:complexType>
                           </xs:element>
                        </xs:sequence>
                     </xs:complexType>
                  </xs:element>
               </xs:choice>
            </xs:complexType>
         </xs:element>
      </xs:choice>
   </xs:complexType>
</xs:element>

<!-- Object Property -->
<xs:element name="oproperty">
   <xs:complexType>
      <xs:choice>
         <xs:element ref="tns:resource"/>
         <xs:element name="union">
            <xs:complexType>
               <xs:sequence>
```

```
            <xs:element ref="tns:oproperty" minOccurs="2"
                        maxOccurs="unbounded"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
   <xs:element name="intersection">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="tns:oproperty" minOccurs="2"
                        maxOccurs="unbounded"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
   <xs:element name="difference">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="tns:oproperty" minOccurs="2"
                        maxOccurs="unbounded"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
   <xs:element name="composition">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="tns:oproperty" minOccurs="2"
                        maxOccurs="unbounded"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
   <xs:element name="inverse">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="tns:oproperty"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
   <xs:element name="transitive">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="tns:oproperty"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
   <xs:element name="existpred">
      <xs:complexType>
         <xs:choice>
            <xs:element name="onOProperties">
               <xs:complexType>
                  <xs:sequence>
```

```
                        <xs:element ref="tns:oproperty"
                                    minOccurs="2"
                                    maxOccurs="2"/>
                        <xs:element name="predicate"
                                    type="tns:BPredicateType"/>
                     </xs:sequence>
                  </xs:complexType>
               </xs:element>
               <xs:element name="onDProperties">
                  <xs:complexType>
                     <xs:sequence>
                        <xs:element ref="tns:dproperty"
                                    minOccurs="2"
                                    maxOccurs="2"/>
                        <xs:element name="predicate"
                                    type="tns:BPredicateType"/>
                     </xs:sequence>
                  </xs:complexType>
               </xs:element>
            </xs:choice>
         </xs:complexType>
      </xs:element>
      <xs:element name="restrict">
         <xs:complexType>
            <xs:sequence>
               <xs:element ref="tns:oproperty"/>
               <xs:element name="domain" minOccurs="0">
                  <xs:complexType>
                     <xs:sequence>
                        <xs:element ref="tns:class"/>
                     </xs:sequence>
                  </xs:complexType>
               </xs:element>
               <xs:element name="range" minOccurs="0">
                  <xs:complexType>
                     <xs:sequence>
                        <xs:element ref="tns:class"/>
                     </xs:sequence>
                  </xs:complexType>
               </xs:element>
            </xs:sequence>
         </xs:complexType>
      </xs:element>
   </xs:choice>
</xs:complexType>
</xs:element>

<!-- Datatype Property -->
<xs:element name="dproperty">
```

```
      <xs:complexType>
         <xs:choice>
            <xs:element ref="tns:resource"/>
            <xs:element name="union">
               <xs:complexType>
                  <xs:sequence>
                     <xs:element ref="tns:dproperty" minOccurs="2"
                                 maxOccurs="unbounded"/>
                  </xs:sequence>
               </xs:complexType>
            </xs:element>
            <xs:element name="intersection">
               <xs:complexType>
                  <xs:sequence>
                     <xs:element ref="tns:dproperty" minOccurs="2"
                                 maxOccurs="unbounded"/>
                  </xs:sequence>
               </xs:complexType>
            </xs:element>
            <xs:element name="difference">
               <xs:complexType>
                  <xs:sequence>
                     <xs:element ref="tns:dproperty" minOccurs="2"
                                 maxOccurs="unbounded"/>
                  </xs:sequence>
               </xs:complexType>
            </xs:element>
            <xs:element name="composition">
               <xs:complexType>
                  <xs:sequence>
                     <xs:element ref="tns:oproperty"
                                 maxOccurs="unbounded"/>
                     <xs:element ref="tns:dproperty"/>
                  </xs:sequence>
               </xs:complexType>
            </xs:element>
            <xs:element name="transform">
               <xs:complexType>
                  <xs:sequence>
                     <xs:element ref="tns:dproperty"/>
                  </xs:sequence>
                  <xs:attribute ref="tns:to" use="required"/>
               </xs:complexType>
            </xs:element>
            <xs:element name="restrict">
               <xs:complexType>
                  <xs:sequence>
                     <xs:element ref="tns:dproperty"/>
                     <xs:element name="domain" minOccurs="0">
```

```
                     <xs:complexType>
                        <xs:sequence>
                           <xs:element ref="tns:class"/>
                        </xs:sequence>
                     </xs:complexType>
                  </xs:element>
                  <xs:element name="range" minOccurs="0">
                     <xs:complexType>
                        <xs:sequence>
                           <xs:element ref="tns:datatype"/>
                        </xs:sequence>
                     </xs:complexType>
                  </xs:element>
               </xs:sequence>
            </xs:complexType>
         </xs:element>
      </xs:choice>
   </xs:complexType>
</xs:element>

<!-- Value -->
<xs:element name="value">
   <xs:complexType>
      <xs:choice>
         <xs:element ref="tns:individual"/>
         <xs:element ref="tns:data"/>
      </xs:choice>
   </xs:complexType>
</xs:element>

<!-- Individual -->
<xs:element name="individual">
   <xs:complexType>
      <xs:sequence>
         <xs:element ref="tns:resource"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>

<!-- Data Value -->
<xs:element name="data">
   <xs:complexType>
      <xs:simpleContent>
         <xs:extension base="xs:string">
            <xs:attribute ref="tns:type" use="required"/>
         </xs:extension>
      </xs:simpleContent>
   </xs:complexType>
</xs:element>
```

```
<!-- Condition -->
<xs:element name="condition">
   <xs:complexType>
      <xs:choice>
         <xs:element name="basic" type="tns:UPredicateType"/>
         <xs:element name="and">
            <xs:complexType>
               <xs:sequence>
                  <xs:element ref="tns:condition" minOccurs="2"
                              maxOccurs="unbounded"/>
               </xs:sequence>
            </xs:complexType>
         </xs:element>
         <xs:element name="or">
            <xs:complexType>
               <xs:sequence>
                  <xs:element ref="tns:condition" minOccurs="2"
                              maxOccurs="unbounded"/>
               </xs:sequence>
            </xs:complexType>
         </xs:element>
         <xs:element name="not">
            <xs:complexType>
               <xs:sequence>
                  <xs:element ref="tns:condition"/>
               </xs:sequence>
            </xs:complexType>
         </xs:element>
      </xs:choice>
   </xs:complexType>
</xs:element>

<!-- Formalism -->
<xs:element name="formalism">
   <xs:simpleType>
      <xs:restriction base="xs:string">
         <xs:enumeration value="GAV"/>
         <xs:enumeration value="LAV"/>
         <xs:enumeration value="GLAV"/>
      </xs:restriction>
   </xs:simpleType>
</xs:element>

<!-- Relation -->
<xs:element name="relation">
   <xs:simpleType>
      <xs:restriction base="xs:string">
         <xs:enumeration value="EQUIVALENT"/>
         <xs:enumeration value="SUBSUMES"/>
         <xs:enumeration value="SUBSUMED"/>
```

```
         <xs:enumeration value="UNSPECIFIED"/>
      </xs:restriction>
   </xs:simpleType>
</xs:element>

<!-- Relational Operator -->
<xs:element name="operator">
   <xs:simpleType>
      <xs:restriction base="xs:string">
         <xs:enumeration value="EQUAL"/>
         <xs:enumeration value="NOT_EQUAL"/>
         <xs:enumeration value="GREATER"/>
         <xs:enumeration value="GREATER_EQUAL"/>
         <xs:enumeration value="LESS"/>
         <xs:enumeration value="LESS_EQUAL"/>
      </xs:restriction>
   </xs:simpleType>
</xs:element>

<!-- Datatype -->
<xs:element name="datatype">
   <xs:complexType>
      <xs:sequence>
         <xs:element ref="tns:condition" minOccurs="0"/>
      </xs:sequence>
      <xs:attribute ref="tns:base"/>
   </xs:complexType>
</xs:element>

<!-- Resource -->
<xs:element name="resource">
   <xs:complexType>
      <xs:attribute ref="tns:uri" use="required"/>
   </xs:complexType>
</xs:element>

<xs:attribute name="uri" type="xs:anyURI"/>
<xs:attribute name="base" type="tns:BaseDatatype"/>
<xs:attribute name="type" type="tns:BaseDatatype"/>
<xs:attribute name="to" type="tns:BaseDatatype"/>



<!--****************************************
*               Useful Types               *
****************************************-->

<!-- Binary Predicate -->
<xs:complexType name="BPredicateType">
```

```
      <xs:sequence>
         <xs:element ref="tns:operator"/>
      </xs:sequence>
   </xs:complexType>

   <!-- Unary Predicate -->
   <xs:complexType name="UPredicateType">
      <xs:sequence>
         <xs:element ref="tns:operator"/>
         <xs:element ref="tns:value"/>
      </xs:sequence>
   </xs:complexType>

   <!-- Datatype -->
   <xs:simpleType name="BaseDatatype">
      <xs:restriction base="xs:string">
         <xs:enumeration value="INTEGER"/>
         <xs:enumeration value="DECIMAL"/>
         <xs:enumeration value="FLOAT"/>
         <xs:enumeration value="DOUBLE"/>
         <xs:enumeration value="STRING"/>
         <xs:enumeration value="BOOLEAN"/>
         <xs:enumeration value="DATETIME"/>
      </xs:restriction>
   </xs:simpleType>
</xs:schema>
```

Appendix C

Mapping Language: Example

This appendix provides a total mapping representation example for the data integration scenario presented in Example 4.1. In this example several GAV type mappings were identified and are presented here in XML format adopting the XML Schema of the Appendix B.

```
<model xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns="http://www.music.tuc.gr/sparqlrw"
       xmlns:ns="http://www.music.tuc.gr/sparqlrw">
   <formalism>GAV</formalism>
   <global>
      <ontology ns:uri="G">
         <name>Mediator Ontology</name>
         <description>The Mediator Ontology</description>
      </ontology>
   </global>
   <locals>
      <ontology ns:uri="S1">
         <name>Source Ontology 1</name>
         <description>The Source Ontology 1</description>
      </ontology>
      <ontology ns:uri="S2">
         <name>Source Ontology 2</name>
         <description>The Source Ontology 2</description>
      </ontology>
   </locals>
   <mappings>
      <cmapping ns:uri="mapping:rule:a">
         <expr1>
            <class>
               <resource ns:uri="G:Science"/>
            </class>
         </expr1>
         <expr2>
            <class>
```

```
            <union>
               <class>
                  <resource ns:uri="S1:Mathematics"/>
               </class>
               <class>
                  <difference>
                     <class>
                        <resource ns:uri="S2:Textbook"/>
                     </class>
                     <class>
                        <resource ns:uri="S2:Novel"/>
                     </class>
                  </difference>
               </class>
            </union>
         </class>
      </expr2>
      <relation>EQUIVALENT</relation>
   </cmapping>
   <cmapping ns:uri="mapping:rule:b">
      <expr1>
         <class>
            <resource ns:uri="G:Autobiography"/>
         </class>
      </expr1>
      <expr2>
         <class>
            <intersection>
               <class>
                  <resource ns:uri="S1:Biography"/>
               </class>
               <class>
                  <existpred>
                     <onOProperties>
                        <oproperty>
                           <resource ns:uri="S1:creator"/>
                        </oproperty>
                        <oproperty>
                           <resource ns:uri="S1:topic"/>
                        </oproperty>
                        <predicate>
                           <operator>EQUAL</operator>
                        </predicate>
                     </onOProperties>
                  </existpred>
               </class>
            </intersection>
         </class>
      </expr2>
      <relation>EQUIVALENT</relation>
```

```
      </cmapping>
      <cmapping ns:uri="mapping:rule:c">
         <expr1>
            <class>
               <resource ns:uri="G:ShortFilm"/>
            </class>
         </expr1>
         <expr2>
            <class>
               <existquant>
                  <onDProperty>
                     <dproperty>
                        <resource ns:uri="S2:runtime"/>
                     </dproperty>
                     <quantifier>
                        <datatype>
                           <condition>
                              <basic>
                                 <operator>LESS_EQUAL</operator>
                                 <value>
                                    <data ns:type="INTEGER">40</data>
                                 </value>
                              </basic>
                           </condition>
                        </datatype>
                     </quantifier>
                  </onDProperty>
               </existquant>
            </class>
         </expr2>
         <relation>EQUIVALENT</relation>
      </cmapping>
      <cmapping ns:uri="mapping:rule:d">
         <expr1>
            <class>
               <resource ns:uri="G:CrossGenre"/>
            </class>
         </expr1>
         <expr2>
            <class>
               <cardinality>
                  <onOProperty>
                     <oproperty>
                        <resource ns:uri="S2:genre"/>
                     </oproperty>
                     <predicate>
                        <operator>GREATER_EQUAL</operator>
                        <value>
                           <data ns:type="INTEGER">2</data>
                        </value>
```

```
                  </predicate>
               </onOProperty>
            </cardinality>
         </class>
      </expr2>
      <relation>EQUIVALENT</relation>
   </cmapping>
   <opmapping ns:uri="mapping:rule:e">
      <expr1>
         <oproperty>
            <resource ns:uri="G:longerThan"/>
         </oproperty>
      </expr1>
      <expr2>
         <oproperty>
            <existpred>
               <onDProperties>
                  <dproperty>
                     <resource ns:uri="S2:runtime"/>
                  </dproperty>
                  <dproperty>
                     <resource ns:uri="S2:runtime"/>
                  </dproperty>
                  <predicate>
                     <operator>GREATER</operator>
                  </predicate>
               </onDProperties>
            </existpred>
         </oproperty>
      </expr2>
      <relation>EQUIVALENT</relation>
   </opmapping>
   <dpmapping ns:uri="mapping:rule:f">
      <expr1>
         <dproperty>
            <resource ns:uri="G:name"/>
         </dproperty>
      </expr1>
      <expr2>
         <dproperty>
            <union>
               <dproperty>
                  <resource ns:uri="S1:title"/>
               </dproperty>
               <dproperty>
                  <resource ns:uri="S2:label"/>
               </dproperty>
            </union>
         </dproperty>
```

```
         </expr2>
         <relation>SUBSUMES</relation>
      </dpmapping>
      <dpmapping ns:uri="mapping:rule:g">
         <expr1>
            <dproperty>
               <resource ns:uri="G:author"/>
            </dproperty>
         </expr1>
         <expr2>
            <dproperty>
               <composition>
                  <oproperty>
                     <resource ns:uri="S1:creator"/>
                  </oproperty>
                  <dproperty>
                     <resource ns:uri="S1:fullname"/>
                  </dproperty>
               </composition>
            </dproperty>
         </expr2>
         <relation>SUBSUMES</relation>
      </dpmapping>
   </mappings>
</model>
```