Computer Science Department
University of Crete

Querying and Updating RDF/S Named Graphs

Master's Thesis

Pediaditis Panagiotis

December 2008
Heraklion, Greece

Querying and Updating RDF/S Named Graphs

Panagiotis Pediaditis
Master's Thesis
Computer Science Department, University of Crete

Abstract

The Semantic Web is emerging to add structure and meaning to conventional Web content [3]. Many scientific groups benefit from data models of increasing complexity to share their findings on a global scale. The Resource Description Framework (RDF) [12] and its vocabulary language RDF-Schema [6] are a popular framework for the development of such data models as it provides the flexibility that is crucial in the ever evolving process of scientific research. A key component to the success of such systems is the ability to manage the data models through query and update mechanisms. Many languages have emerged to facilitate such functionality, the most dominant of these being SPARQL [28].

In the creation of large dynamic and distributed repositories, the need to assess the credibility of the information available is inherent. This is more commonly known as the problem of *Trust on the Web* which is dealt with by associating information on the Semantic Web with its origin also called *Provenance*. One of the recent extensions of RDF namely *Named Graphs* [9] provides the means to modularize and reify RDF/S graphs, a key functionality in enabling the recording and tracking of provenance on the Semantic Web.

However, named graphs have no support for descriptions utilizing schema information. The goal of this thesis is to integrate the functionality of named graphs to systems that are schema aware. Distribution of schema information among different sources can lead to inferred information originating from more than one source. This mandates the introduction of a new construct, the *Graphset*. The proposed construct can describe the origin of a piece of information whether it is a single named graph or multiple named graphs.

This master thesis also describes the semantics of querying and updating graphsets. Moreover, the RDF database system found in RDF-Suite [1] as well as RQL [19] and RUL [23] languages have been extended to support the new construct of graphsets.

Supervisor: Vassilis Christophides
Associate Professor

Επερώτηση και Ενημέρωση RDF/S Ονοματισμένων Γράφων

Πεδιαδίτης Παναγιώτης
Μεταπτυχιακή Εργασία
Τμήμα Επιστήμης Υπολογιστών, Πανεπιστήμιο Κρήτης

Περίληψη

Ο Σημασιολογικός Ιστός αναδύεται σαν μια επέκταση του Παγκόσμιου Ιστού που στοχεύει στο να προσθέσει δομή στο υπάρχον συμβατικό περιεχόμενο [3]. Πολλές επιστημονικές ομάδες αξιοποιούν μοντέλα δεδομένων αυξανόμενης πολυπλοκότητας για τον διαμοιρασμό των ευρημάτων τους σε παγκόσμια εμβέλεια. Η γλώσσα περιγραφής πόρων RDF [12] καθώς και η γλώσσα περιγραφής σχημάτων RDF-Schema [6] αποτελούν μια δημοφιλή πλατφόρμα ανάπτυξης τέτοιων μοντέλων δεδομένων καθώς παρέχουν ευελιξία η οποία είναι κρίσιμη στην συνεχώς εξελισσόμενη διεργασία της επιστημονικής έρευνας. Ένα από τα κύρια συστατικά στην επιτυχία τέτοιων συστημάτων είναι η δυνατότητα διαχείρισης του περιεχομένου μέσω μηχανισμών επερώτησης και ενημέρωσης.

Η δημιουργία μεγάλων, δυναμικών και διανεμημένων βάσεων γνώσης οδηγεί στην ανάγκη αξιολόγησης της αξιοπιστίας των διαθέσιμων πληροφοριών. Αυτό είναι ευρέως γνωστό ως το πρόβλημα *Trust on the Web* λύση του οποίου αποτελεί η συσχέτιση των πληροφοριών με την προέλευση τους *Provenance* . Μία από τις πρόσφατες επεκτάσεις της RDF , οι *ονοματισμένοι γράφοι* [9], επιτρέπουν την ομαδοποίηση και αναφορά σε RDF γράφους. Αυτή η λειτουργικότητα είναι κλειδί στην καταγραφή και ανίχνευση της προέλευσης στον Σημασιολογικό Ιστο.

Οι ονοματισμένοι γράφοι δεν υποστηρίζουν περιγραφές δεδομένων οι οποίες συνοδεύονται από πληροφορία σχήματος. Ο στόχος αυτής της μεταπτυχιακής εργασίας είναι η ολοκλήρωση της λειτουργικότητας των ονοματισμένων γράφων σε βάσεις γνώσεις που έχουν σχήμα. Η εξαγωγή συμπερασμάτων μέσω σχήματος μπορεί να οδηγήσει σε πληροφορίες που προέρχονται από πολλαπλές πηγές. Αυτό απαιτεί την εισαγωγή μιας καινούριας δομής, τα σύνολα γράφων (*Graphsets*) . Η δομή αυτή επιτρέπει την καταγραφή της προέλευσης μιας πληροφορίας είτε αυτή είναι ένας είτε πολλαπλοί γράφοι.

Επίσης, η εργασία αυτή περιγράφει την σημασιολογία γύρω από την επερώτηση και ενημέρωση των Graphsets . Παραπέρα, η βάση αποθήκευσης RDF δεδομένων της RDF-Suite [1], καθώς και οι αντίστοιχες γλώσσες επερώτησης και ενημέρωσης, RQL [19] και RUL [23] επεκτάθηκαν έτσι ώστε να υποστηρίζουν την νέα προτεινόμενη λειτουργικότητα.

Επόπτης Καθηγητής: Βασίλης Χριστοφίδης

Αναπληρωτής Καθηγητής

Ευχαριστίες

Αρχικά θα ήθελα να ευχαριστήσω θερμά τον επόπτη καθηγητή μου κ. Βασίλη Χριστοφίδη ο οποίος υπήρξε πολύτιμος καθοδηγητής στα ακαδημαϊκά μου χρόνια και η κινητήρια δύναμη πίσω από αυτήν την εργασία. Επίσης οφείλω ένα μεγάλο ευχαριστώ στην Ειρήνη Φουντουλάκη και τον Γιώργο Φλουρή οι οποίοι συνέβαλαν σε τεράστιο βαθμό σε αυτήν την δουλειά.

Θέλω ακόμα να πω ευχαριστώ σε όλα τα μέλη της ομάδας μου για την ευχάριστη συνεργασία τα τελευταία χρόνια. Ακόμα περισσότερο, θέλω να ευχαριστήσω τους καλούς μου φίλους στο Ηράκλειο που έκαναν τα χρόνια αυτά να αξίζουν.

Τέλος, και πάνω από όλα, θέλω να ευχαριστήσω τους γονείς μου Μανώλη και Ίνγκα και τις αδερφές μου, Κάλλια και Έμιλυ που με την αγάπη και υποστήριξη τους μπορώ και κυνηγάω τα όνειρα μου.

Contents

1 Introduction **1**

2 Manipulating RDF/S Named Graphs **7**

2.1 Motivating Example 7

2.2 Querying RDF/S Named Graphs 10

2.3 Updating RDF/S Named Graphs 12

2.4 Molding it all together 13

3 Resource Description Framework **17**

3.1 RDF and RDF Schema 17

3.2 Formalizing RDF 20

3.2.1 RDF Graphs 20

3.2.2 RDFS Graphs 20

3.2.3 RDFS Inference 21

3.2.4 Typing 21

3.2.5 Validity 24

4 Named Graphs and Graphsets **29**

4.1 Named Graphs 29

4.2 RDF/S GraphSets 31

4.3 Reasoning on RDF/S Graphsets 34

4.3.1 Inference 35

4.3.2 Validity 36

4.3.3 Redundancy Elimination 38

4.3.4 Complexity Analysis 38

5 Managing Extended RDF/S graphs **43**

5.1 Querying Extended RDF/S Graphs 43

5.2 Updating Extended RDF/S Graphs . . . 45
5.2.1 INSERT Operation . . . 45
5.2.2 DELETE Operation . . . 47

6 RUQL with RDF/S Named Graphs 53
6.1 RQL with Graphsets . . . 53
6.1.1 Subclass and Subproperty Transitivity and Type Inference . . . 54
6.1.2 Path Expressions . . . 54
6.2 Declaring and using named graphs and graphsets . . . 55
6.2.1 USING NAMEDGRAPH clause . . . 55
6.2.2 Graphset Operations . . . 56
6.2.3 The DEFAULT# Graphset . . . 56
6.3 RUL with Graphsets . . . 56
6.3.1 Class instance insertion to a graphset . . . 57
6.3.2 Class instance deletion from graphset . . . 59
6.3.3 Property instance insertion to graphsets . . . 61
6.3.4 Property instance deletion . . . 63
6.4 REPLACE operation . . . 64

7 Implementation 65
7.1 Relational Model . . . 65
7.1.1 RDF Schema Representation . . . 65
7.1.2 RDF Data Representation . . . 66
7.1.3 Graphset Representation . . . 67
7.2 Extending the RQL Interpreter . . . 67
7.2.1 Syntax . . . 69
7.2.2 Graph Construction . . . 69
7.2.3 Type Inference . . . 70
7.2.4 Basic Database Functions . . . 70
7.2.5 Execution Graph Evaluation . . . 73
7.2.6 Database API . . . 73

8 Related Work 77
8.1 SPARQL . . . 77
8.2 SPARQL/Update . . . 78
8.3 TriQL . . . 79

8.4 TRIPLE 79
8.5 Other Languages 80
8.6 Named Graph Extentions 80

9 Conclusions & Future Work **81**

References **82**

A The BNF grammar of RUL **87**

List of Figures

1.1 The layered architecture of the Semantic Web [41]. 2
1.2 Collaborative Neurobiology Ontology, motivating example. 4

2.1 Schema provided by IUPHAR 8
2.2 Schema and data provided by Pfeizer 8
2.3 Schema provided by GSK 9
2.4 "Integrated" Biology Schema 10

4.1 Graphset Example 33
4.2 Collaborative Neurobiology Ontology 36
4.3 sc Hierarchy 40

5.1 Class Instance Insertion 46
5.2 Class Instance Deletion 48
5.3 Class Instance Deletion Effects and Side-effects 49
5.4 Property Instance Deletion Effects and Side-effects 51

6.1 Example class instance insertion effects and sideffects. 58
6.2 Example class instance deletion effects and sideffects. 60
6.3 More example class instance deletion effects and sideffects. 62
6.4 Example property deletion effects and sideffects 63

7.1 Relations used for schema representation. 66
7.2 Relations used for the representation of data. 67
7.3 Relations used for the representation named graphs, graphsets and their association with triples. 67
7.4 Overview of the architecture of the RQL interpreter and its extention for named graphs. 68

List of Tables

3.1 Part of RDFS Inference Rules 21
3.2 Basic Inference Rules 24
3.3 Typing Rules 25
3.4 Default Typing Rules 26
3.5 Conflict Free Validity Rules 26
3.6 Completeness Validity Rules 27

4.1 Part of RDFS Inference Rules with Graphsets 35
4.2 Conflict Free Validity Rules 37
4.3 Completeness Validity Rules 38
4.4 Redundancy Elimination Rules 39

6.1 RQL Path elements 55
6.2 Composing path elements 55

Chapter 1

Introduction

The Semantic Web is emerging to add structure and meaning to conventional Web content [3]. This is achieved by enhancing the existing data (usually in HTML or XML format) with meta-data (using RDF, OWL and others) which specify the meaning of data. Subsequently, Web based applications are empowered with the ability to perform intelligent reasoning on the existing information. An overview of the architectural overview of the Semantic Web as visualised by the W3C can be seen in Figure 1.1.

The ability to *query*, and in some applications *update*, the meta-data made available on the Semantic Web is vital to any system that takes advantage of their power. As a result, a number of query languages and platforms have emerged, enabling the query and update of semantic data, predominantly RDF. The query process is less trivial than the conventional Web as proper reasoning must be performed on the information to reach meaningful conclusions.

The Resource Description Framework (RDF) [12] is presented as a general-purpose language for representing information in the Web. It allows the description of Web resources that provides for the desired interoperability between applications. RDF-Schema [6] is a vocabulary description language built on top of RDF that enables the declaration of *classes* and *properties* as well as the relationships between *classes* and *properties*. The specification of RDFS contains some basic inference rules which, when applied to an ontology with schema information, can provide us with information that is entailed by the ontology but is not explicitly provided. RDFS reasoning is in some cases performed separately from querying and in others is built into the query mechanism.

A number of RDF query languages have support for RDFS reasoning. RQL [19] makes full use of RDFS inference rules utilizing an intelligent labeling technique which encodes transitive IsA relations between classes as well as properties. SeRQL [7] is a language,

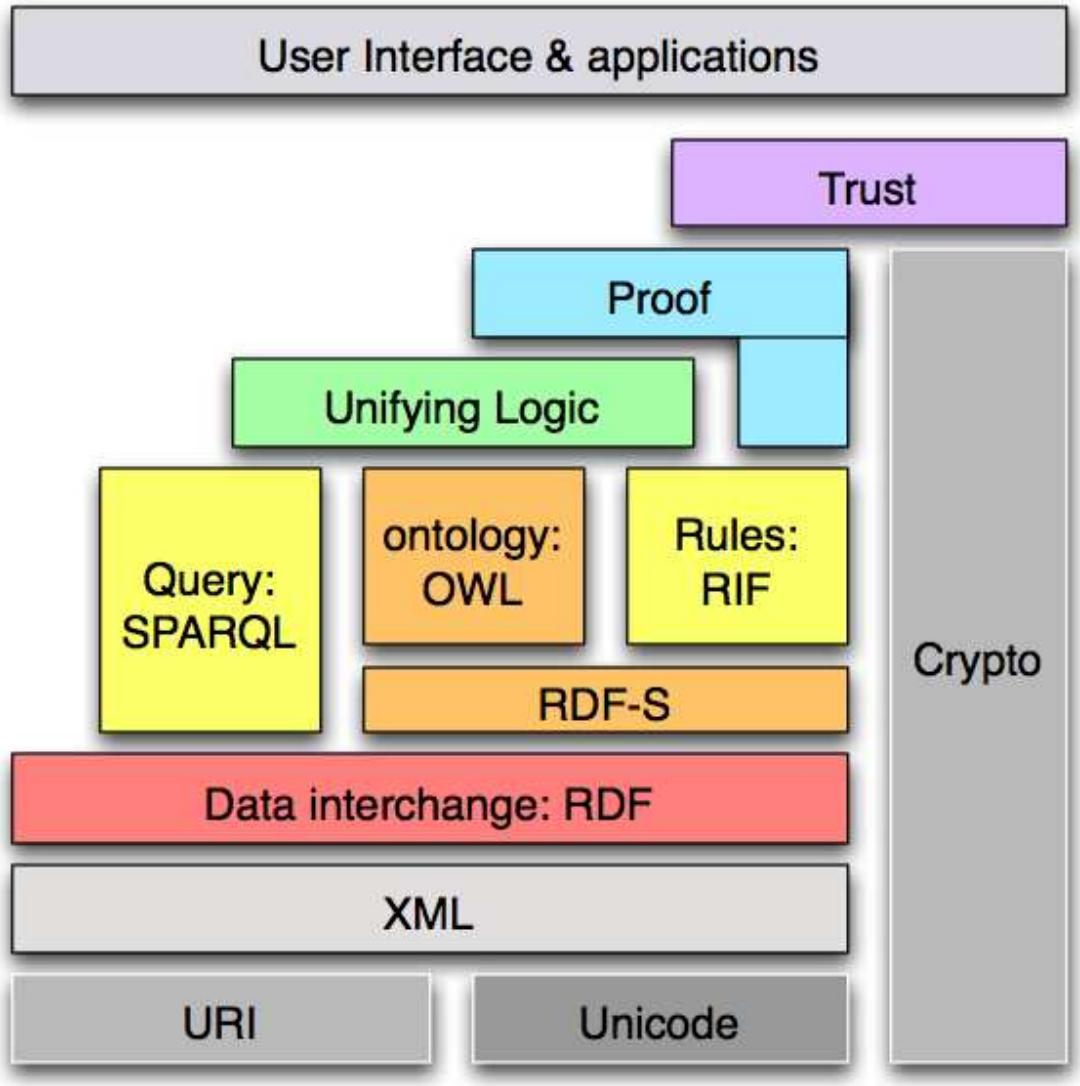


Figure 1.1: The layered architecture of the Semantic Web [41].

syntactically similar to RQL, that also supports RDF Schema inference. Other logic based languages such as TRIPLE [38] can indirectly support querying implicit information, however, they require the definition of the respective RDFS rules. Such approaches have a significant overhead when querying large hierarchies. There is only one language up to date that supports *updates* in RDF/S graphs (i.e. RDF graphs with implicit information), RUL [23]. RUL is an extention of RQL and supports updates at a data level and has build in sideffect operations that maintain the correctness of the RDF/S graph.

The Semantic Web is a large and dynamic repository in which anyone can create and distribute information. This naturally introduces the question: *How credible is a piece of information?* This is more commonly known as the problem of *Trust on the Web* and deals

with the issue of determining the credibility of web information. Trust is placed at the top layers of the Semantic Web stack as seen in Figure 1.1. This is achieved by associating information on the Semantic Web with its origin also called *Provenance*. However, RDF on its own is incapable of capturing provenance in an efficient way. Therefore, *named graphs* was suggested by [9] as an extention over RDF. This mechanism essentially enabled the reification of entire RDF graphs with which provenance information could be associated. Query languages such as SPARQL[28] incorporated named graphs into their functionality.

To establish the importance of both RDFS reasoning and RDF graph reification, consider an e-science application in which biologists from different organisations are collaboratively developing and using an ontology with biological data. In general, provenance information concerning the creation, attribution or version history of biological data is crucial for assessing its integrity and scientific value. Provenance information summarizes among others the origin of data.

Since the information added to the ontology we will study comes from many different sources, it is important to be able to effectively track where this information originates from, or in other words, track its provenance. With provenance, the biologists of each organization can judge whether a piece of information is credible or not based on who its creators are. Consider the example in Figure 1.2. In RDF, all information is captured by the arcs in an ontology graph. In our example we visualise how information originates from different sources by colouring the arcs. For example the information that *dopamine* is a *neuromodulator* is provided by a medical center called *HADASSAH* (4). On the other hand, the schema information that all *neuromodulators* are *neurotransmitters* is provided by the organisation *IUPHAR* (2). The predominant technique for adding provenance information to RDF triples (graph arcs), is the use of *named graphs*. Essentially, all triples originating from a specific source are grouped under a named graph which is associated with that source. More often than not, a lot of information in an RDF graph is not explicitly stated in triples but can be inferred through reasoning. For example, though not stated directly, from the information in Figure 1.2 we can deduce that *dopamine* is also a *neurotransmitter* because the schema states that all *neuromodulators* are *neurotransmitters*. The process of reasoning using basic rules, such as those defined by RDFS, is common in many RDF frameworks as mentioned earlier. However, with the introduction of named graphs for provenance, the semantics of reasoning whilst retaining provenance information has yet to be explored. Consider our example; the source of the information that *dopamine* is a *molecule* can not be directly decided. In this case there are more than one sources that contribute to the information. In terms of provenance we view the origin

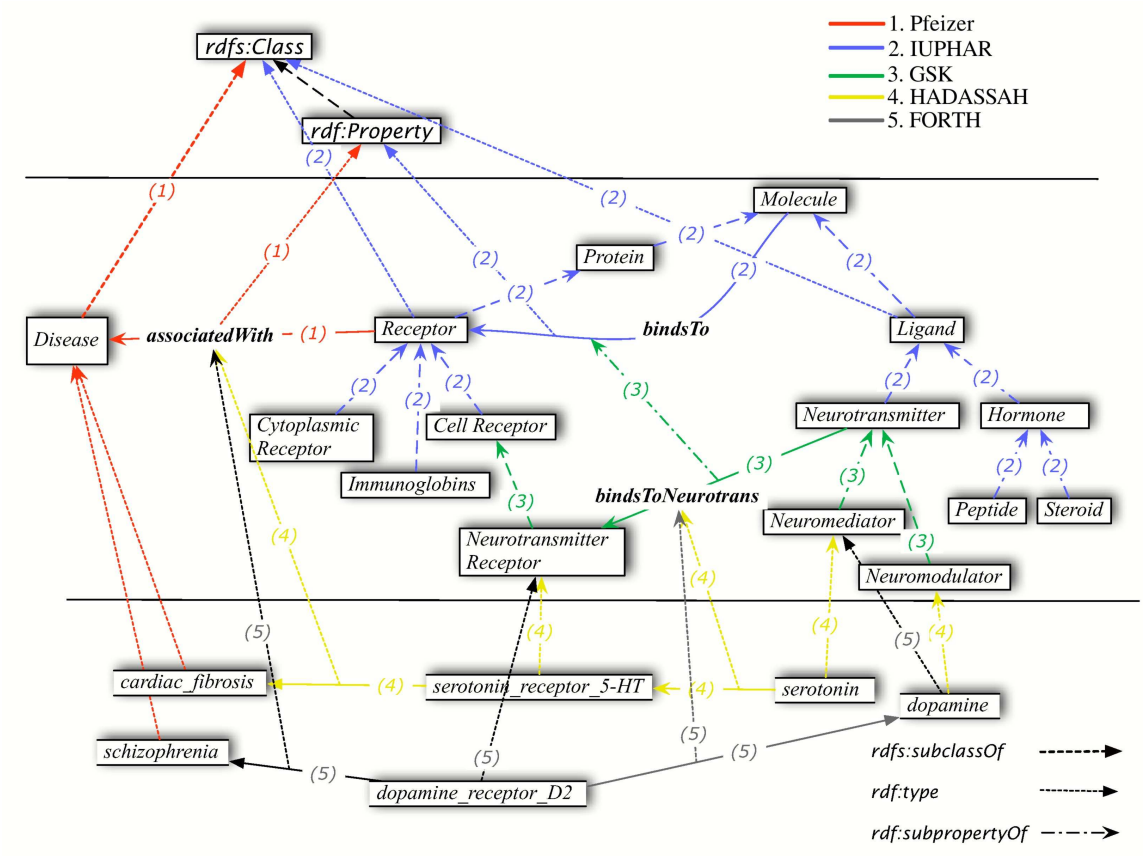


Figure 1.2: Collaborative Neurobiology Ontology, motivating example.

of the information as *composite* (i.e. being shared by two different sources). However, the original RDF construct, named graphs is unable to capture the notion of composite origin. This issue becomes more so apparent in the presence of updates, where inferred information may become explicit as a result of a deletion but its provenance information must be maintained. To overcome this, we introduce a new concept, the *graphset*, which allows the association of RDF/S graphs with sets of sources (named graphs) instead of individual ones.

The main contribution of this work is the definition of a formal model for RDF that can support named graphs and provenance recording in ontologies that also use RDF Schema and are aware of reasoning and inferred information. Moreover, we present the formal semantics of querying and updating RDF descriptions in such ontologies. This includes the effects and sideffects of inserting data to or deleting it from an RDF named graph. Finally, as a proof of concept, we extend a known RDF query and update language RUL to support the recording of provenance. In contrast to other languages, RUL has the advantage of already supporting queries and data updates that are schema aware.

The structure of this thesis is as follows: In Chapter 2 we present a series of examples

motivating the need to track provenance in the presence of schema information. At the same time we will be informaly introducing a syntax for, and the semantics of querying and updating named graphs. In Chapter 3 we briefly describe the Resource Description Framework and its vocabulary description language RDFS subsequently we present a formal theoretical model for RDF and RDFS. Chapter 4 fully integrates the construct of named graphs into our formal model and formally introduces the notion of graphsets, our answer to the issue of shared ownership of triples along with extended RDF/S graphs, the extention of RDF graphs that supports graphsets. Chapter 5 presents the semantics involved in querying and in updating extended RDF/S graphs. Chapter 6 presents the syntax and semantics of the RDF query and update language RUL as they have been extended to support graphsets. In Chapter 7 we describe our extention of the RQL interpreter to support querying and updating of of graphsets. In the final chapters we take a look at some related work, gather our conclusions and motivate possible future work.

Chapter 2

Manipulating RDF/S Named Graphs

In this chapter we discuss in length, albeit informally, *named graphs* and how they can be used to capture the *provenance* of information as well as the *query* and *update* languages that will be used to manipulate RDF graphs enriched with named graphs. These languages are extentions over the RDF update and query languages RQL [19] and RUL [23].

2.1 Motivating Example

During the last years we have been witnessing an increase in the interest for, and the use of RDF technologies in the field of e-science and more specifically in biological applications [25, 32]. Biologists are especially keen in using RDF since it gives users the ability to create descriptions in a very flexible and powerful way. A large number of research groups are working (as in most disciplines) independently and reach various experimental conclusions. It is essential for scientists to be able to have access to this huge amount of information, and at the same time track its *provenance*. As mentioned in the previous chapter, recording the provevance of data is crucial to assess its correctness and scientific value.

We drew our inspirations from bioinformatics and biomedical applications and designed an RDF/S schema that captures information related to *diseases*, *receptors*, *ligands* and the *relationships* between them. The information (schema and data) in our biology ontology is contributed by several authorities. In Figure 2.1 we illustrate the classes and properties that we drew from the IUPHAR [1] database [16], a major player in the domain of Basic and Clinical Pharmacology.

[1] http://www.iuphar-db.org/

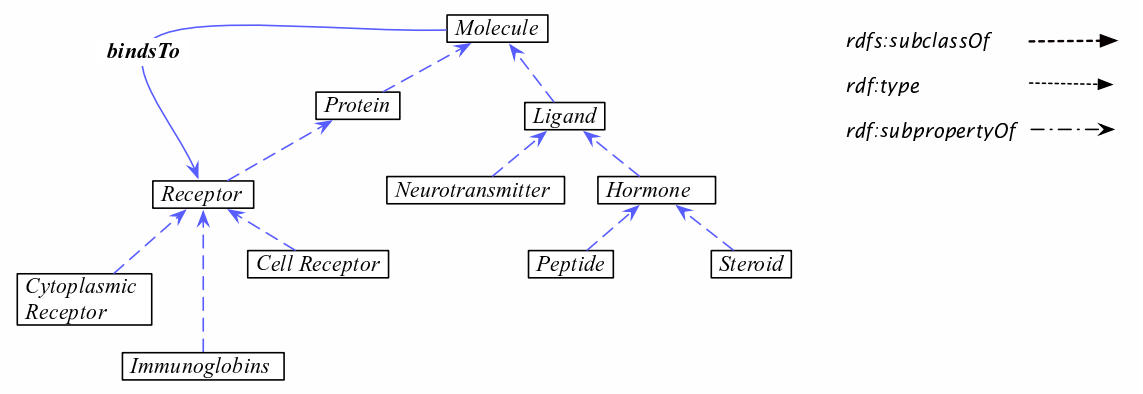


Figure 2.1: Schema provided by IUPHAR

Classes *Molecule* and *Protein* represent all molecules and proteins respectively. The latter is related to the former with a *subclass of* relation which indicates that all proteins are molecules. Classes *Receptor* (represents all receptors) and *Ligand* (represents all ligands) are subclasses of *Protein* and *Molecule* respectively. Relationship *bindsTo* associates a molecule (instance of class *Molecule*) with a receptor (instance of class *Receptor*). Classes *Neurotransmitter* and *Hormone* are subclasses of *Ligand* and represent all neurotransmitters and hormones respectively. Class *Hormone* is further specialized to the classes *Peptide* and *Steroid*. The IUPHAR schema also contains the hierarchy of the different classes of receptors: *Cytoplasmic* and *Cell* receptors as well as *Immunoglobulins*.

In addition to IUPHAR, pharmaceutical company Pfizer[2] defines the classes and instances that are associated with diseases (shown in Figure 2.2). Class *Disease* represents all diseases, and property *associatedWith* connects a receptor (instance of the IUPHAR class *Receptor*) with a disease. In addition, Pfeizer defines the following diseases: `&cardiac_fibrosis` and `&schizophrenia`. These are instances of a *Disease* and are essentialy RDF/S *individuals*.

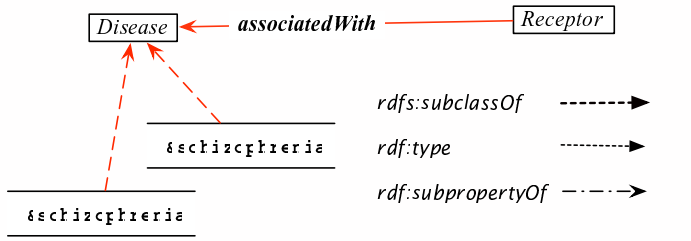


Figure 2.2: Schema and data provided by Pfeizer

[2]Pfizer: http://www.pfizer.com/

Finally, the pharmaceutical company GSK[3] specializes the class *CellReceptor* with the class *NeurotransmitterReceptor* to capture those receptors that modulate signals between neurons and other cells. GSK defines classes *Neuromediator* and *Neuromodulator* as subclasses of *Neurotransmitter*. In addition, property *bindsToNeuroTrans* is a sub-property of *bindsTo* and is defined between classes *Neurotransmitter* and *Neurotransmitter Receptor*. The part of the RDF/S graph contributed by GSK is presented in Figure 2.3. For ease of presentation, we have included the necessary classes defined by IUPHAR (Figure 2.1).

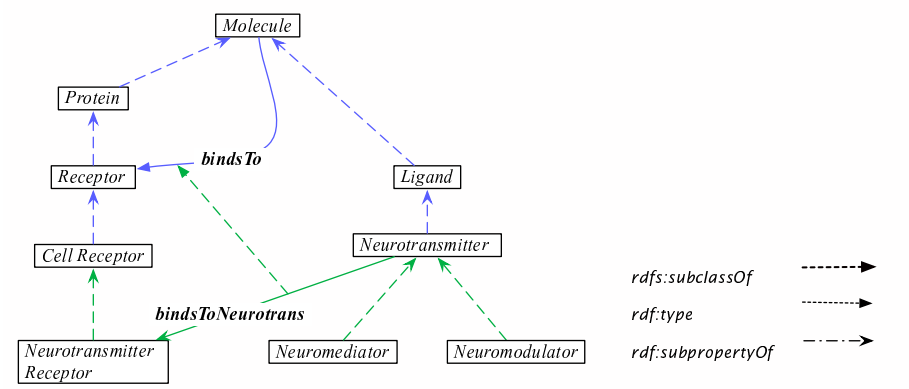


Figure 2.3: Schema provided by GSK

The research center FORTH on Crete has a Biomedical Informatics Laboratory[4] working on creating novel mathematical and computational tools that can be used to solve complex, data intensive biological problems. In our example researchers at FORTH have added some information related to dopamine. More specifically, `&dopamine` and `&dopamine_receptor_D2` are added as instances of the GSK classes *Neuromodulator* and *Neurotransmitter Receptor* resp., and related through a property, instance of *bindsToNeuroTrans*. Moreover, `&dopamine_receptor_D2` is connected to the disease `& schizophrenia` (defined by Pfizer) through an *associatedWith* property. Similar to FORTH, Hadassah[5] has provided serotonin-related information. `&serotonin` and `&serotonin_receptor_5-HT` are added as instances of the GSK classes *Neuromediator* and *Neurotransmitter Receptor* respectively, and related through the property *bindsToNeuroTrans*. Moreover, `&serotonin_receptor_5-HT` is connected to the disease `&cardiac_fibrosis` (defined by Pfizer) through an *associatedWith* property. Finally, FORTH also adds a new type of disease: the individual `&smallpox`.

[3] GlaxoSmithKline: http://www.gsk.com
[4] http://www.ics.forth.gr/bmi/bioinformatics.html
[5] http://www.hadassah.org.il/english

In Figure 2.4 we show the entire RDF/S graph produced by *"integrating"* the graphs discussed earlier. We use *named graphs* in order to distinguish the origin ontology classes and individuals we define one named graph per source separate RDF/S graph, we label each arc in the graph and associate the label with one of the named graphs we consider.

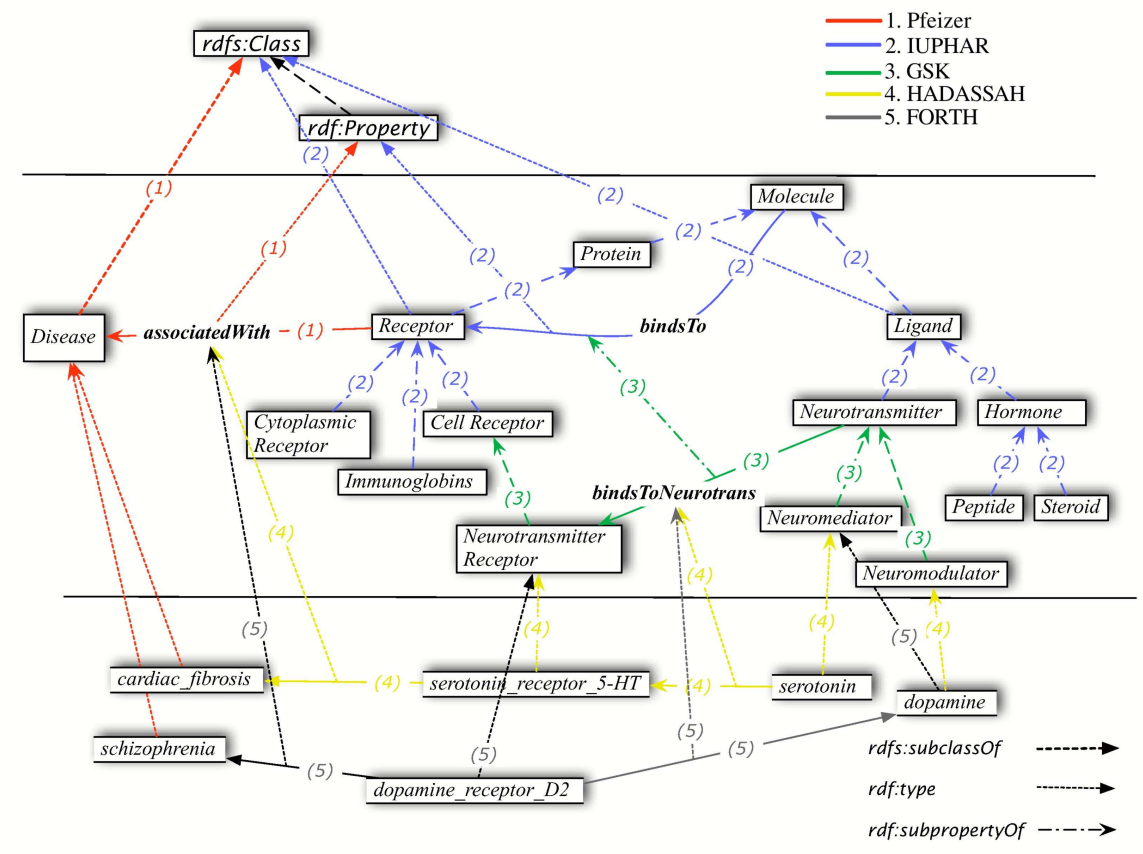


Figure 2.4: "Integrated" Biology Schema

2.2 Querying RDF/S Named Graphs

Consider the following scenario: Jake, a student of pharmacology wants to study some of the diseases, receptors and ligands described in the ontology we presented previously. The data available to the student is illustrated in Figure 2.4. We are going to discuss here a set of queries that Jake would propably ask to the database in order to retrieve information. First, suppose that Jake needs to retrieve information that is associated with a given named graph. For example, suppose that Jake would like to find out *"the diseases that are defined by Pfizer"*. The query that expresses the above request is shown below:

```
Q1: SELECT x FROM g::Disease{x} USING NAMEDGRAPH g=&http://biology#Pfeizer
```

Query Q1 returns the resources &cardiac_fibrosis and &schizophrenia, instances of class *Disease* since these are the only two diseases defined by Pfizer. Notice that &smallpox is also a disease however it is not part of the query result because it is not defined in Pfizer. The pattern Disease{x}, asks for the instances of class *Disease*. Prefixing this pattern with g :: will assign to variable g the named graphs to which the resulting instances are defined. Finally, the USINGNAMEDGRAPH clause restricts the named graph to be the one whose name is &http : //biology#Pfeizer.
Suppose now that Jake would like to explore *the receptors and the diseases they are associated with by Pfizer*. The query that he needs to formulate is:

```
Q2: SELECT x,y FROM g::{x}associatedWith{y}
USING NAMEDGRAPH g=&http://biology#Pfizer
```

Pattern {x}associatedWith{y} binds the pair of variables x, y to those pairs of *resources* connected with a property named associatedWith.
Now our student wants to find out which classes and properties that are defined by the pharmaceutical company *GSK*. To retrieve this information he writes queries Q3 and Q4: this using the following two queries:

```
Q3: SELECT $C FROM g::$C USING NAMEDGRAPH g=&http://biology#GSK
```

```
Q4: SELECT @P FROM g::@P USING NAMEDGRAPH g=&http://biology#GSK
```

In Q3, variable $C ranges over all *classes*. g holds the named graphs of sets of named graphs in which $C is defined as a class type. The same observation holds for query Q4: the difference is that @P now ranges over properties and not classes.
Followingly, Jake wants to find all class instantiations in which GSK participates (i.e. shares ownership of the instance). So he performs the following query:

```
Q5: SELECT x FROM g::$C{x} WHERE g > &http://biology#GSK
```

The operator > suggest that g is a set of named graphs that contains at least the named graph &http://biology#GSK
Through his queries, Jake realized that all *neuromodulators* are *ligands*. To investigate this information further he wishes to track the provenance of the *subclass of* relation established between classes *Neuromodulator* and *Ligand*. To achieve this he formulates query Q6:

```
Q6: SELECT g FROM g::Ligand{;$C} WHERE $C=NeuroModulator
```

The above query will find all subclasses of *Ligand* and keep only the *NeuroModulator* class (pattern `WHERE $C = NeuroModulator`. In this query, variable `g` will hold the graphsets from which the subclass relation between classes *Ligand* and *NeuroModulator*.

The result is the graphset {&`http://biology#GSK`, &`http://biology#IUPHAR`} because the subclass relationship in question is *inferred* from the following *explicit* subclass of relationships: one that states that class *NeuroModulator* is a subclass of class *NeuroTransmitter* (defined in the named graph &`http://biology#GSK`) and a second one that specifies that class *NeuroTransmitter* is a subclass of class *Ligand* (from named graph {&`http://biology#IUPHAR`).

2.3 Updating RDF/S Named Graphs

Our pharmacology student Jake has now performed a number of experiments and wants to add his results to the RDF/S graph. To be able to distinguish his results from those of other collaborators he creates a new named graph and populate the graph with the triples that describe his findings. First he creates his graph with name &`http://biology#jakes_graph` with the following update statement:

```
U1: INSERT Graph(&http://biology#jakes_graph)
```

Update statement `U1` will create a new instance of class *rdfSuite*:*Graph*and assign name *http://biology#jakes_graph* to it.
One of Jake's experiments showed that the chemical compound *acetylcholine is a neuromodulator*. He adds this information to his RDF graph using the update `U2`:

```
U2: INSERT g::NeuroModulator(&acetylcholine)
USING NAMEDGRAPH g=&http://biology#jakes_graph
```

This update associates the information that resource &`acetylcholine` is an instance of the class *NeuroModulator* with the named graph &`http://biology#jakes_graph` defined in the $USINGNAMEDGRAPH$ clause.
Jake also isolated a receptor of *acetylcholine* named *M1* and adds this information to his named graph:

```
U3: INSERT g::NeuroTransReceptor(&M1)
USING NAMEDGRAPH g=&http://biology#jakes_graph
```

Finally he adds the information that *acetylcholine* binds to receptor *M1* through property *bindsToNeuroTrans*:

```
U4: INSERT g::bindsToNeuroTrans(&acetylcholine,&M1)
USING NAMEDGRAPH g=&http://biology#jakes_graph
```

The biologists in Pfizer conducted experiments using antagonists of the *M1* receptor and found that blocking the receptor has an effect on short term memory. So they add *memory loss* as a disease and associate receptor *M1* with it:

```
U5: INSERT g::Disease(&memory_loss), g::associatedWith(&M1,&memory_loss)
USING NAMEDGRAPH g=&http://biology#Pfeizer
```

In one of there experiments, the biologists in Pfizer reached the erroneous conclusion that memory loss is also associated with the receptor *M2*. After discovering their mistake they removed the association with the following update:

```
U5: DELETE g::associatedWith(&M2,&memory_loss)
USING NAMEDGRAPH g=&http://biology#Pfeizer
```

2.4 Molding it all together

In some preliminary work done at HADASSAH medical center, researchers want to find all molecules that bind to receptors that are associated with memory loss in the current RDF/S graph. They formulate query

```
Q7: SELECT x FROM {x}bindsTo{y}.associatedWith{z}
WHERE z=&memory_loss
```

The results of Q7 reports that individual &*acetylcholine* is indirectly associated with *memoryloss*. However, they seem very sceptic about this information and wish to track its origin. So they expand their previous query:

```
Q8: SELECT x,g FROM g::{x}bindsTo{y}.associatedWith{z}
WHERE z=&memory_loss
```

Along with the requested molecules, the results of the above query will contain the named graphs which participated in producing the resulting information. In the case of &`acetylcholine` the named graphs associating it with memory loss are &`http://biology#jakes_graph` and &`http://biology#Pfeizer`.
It so happened that the biologists at FORTH also reached the same conclusion as Pfeizer that receptor *M1* is associated with memory loss. As a result they added the respective property instance to their named graph with the query:

```
U6: INSERT g::associatedWith(&M1,&memory_loss)
USING NAMEDGRAPH g=&http://biology#FORTH
```

Now if the researchers at HADASSAH repeat their last query they will find that the returned result originates from two different sources. Firstly from the named graphs &http://biology#jakes_graph, &http://biology#Pfeizer and secondly from the named graphs &http://biology#jakes_graph,&http://biology#FORTH.
After receiving an earfull from his supervisor, Jake reconsiders his initial notion that &*acetylcholine* is a *neuromodulator*, and infact doesn't believe it is even a *neurotransmitter*. So he deletes &*acetylcholine* as an instance of *NeuroTransmitter* class.

```
U7: DELETE g::NeuroTransmitter(&acetylcholine)
WHERE g=&http://biology#jakes_graph
```

To the student's surprise, the above update statement failed. He realised that the acetylcholine is not declared as an instance of class *Neurotransmitter* in his graph but is shared by the graphset {&http://biology#jakes_graph, &http://biology#GSK} So he revises his update statement:

```
U8: DELETE g::NeuroTransmitter(&acetylcholine)
WHERE g=gset(&http://biology#jakes_graph,&http://biology#GSK)
```

This update results in the reclassification of &acetylcholine as an instance of *Ligand* (parent of class *Neurotransmitter*). However, although the information that &acetylcholine is a *neuromodulator* was provided by Jake alone, the information that &acetylcholine is an instance of *Ligand* belonged to Jake and GSK and IUPHAR representively. So, to maintain the provenance information, the new instantiation of &acetylcholine is associated with the set of named graphs {&http://biology#GSK, &http://biology#jakes_graph, &http://biology#IUPHAR}
Subsequently, since *acetylcholine* is no longer a classified as a neurotransmitter it can no longer participate in the property bindsToNeurotrans. So *acetylcholine's* relation to &M1 is reclassified to bindsTo. Again this property instance used to be inferred through the sp relation between properties *bindsToNeuroTrans* and *bindsTo*. Therefore the graphset to which the explicit now triple is associated with, is the graphset to which it belonged when it was still implicit: {&http://biology#GSK, &http://biology#jakes_graph}.
The head of research at FORTH realized that his team had made a critical error in their experiments that associated receptors with diseases and requested that they be revised. To remove their existing findings they performed the update:

```
U9: DELETE g::associatedWith(x,y)
FROM g::{x}associatedWith{y}
WHERE g=&http://biology#FORTH
```

Note that the update did not, in any way, affect the respective information held in Pfizer's named graph.
Later on, Jake's supervisor decided that he was not fit for the project and had him reassigned. Subsequently, he wanted to remove the entire named graph associated with Jake. To achieve this he performed the statement

```
U10: DELETE Graph(&http://biology#jakes_graph)
```

However this operation failed stating that the named graph `&http://biology#jakes_graph` is not empty. To enforce the deletion of any data shared by Jake's graph along with the graph itself the following update must be used:

```
U11: DELETE Resource(&http://biology#jakes_graph)
```

Chapter 3

Resource Description Framework

In this chapter we will discuss the Resource Description Framework (RDF) [12], a language for representing information about resources in the World Wide Web and RDF Schema [6], an extensible knowledge representation language that provides basic elements for the description of RDF vocabularies intended to structure RDF resources. RDF and RDFS are presented informally in Section 3.1; a more formal discussion on the semantics of RDF and RDFS as we perceive them in our work is presented in Section 3.2, setting the basis for the discussion of *named graphs* that will be discussed in Chapter 4.

3.1 RDF and RDF Schema

The Resource Description Framework, commonly RDF [12], is a framework for representing information about Web resources. It is comprised of W3C recommendations that enable the encoding, exchange and management of structured metadata, allowing therefore the publication of both human-readable and machine-processable descriptions of Web resources. RDF is based on a simple data model that is easy for applications to process and manipulate RDF data. We state here the basic assumptions underlying RDF.

- In RDF essentially anything we wish to describe is a *resource*. A resource may be anything from a person to an institution, the relation a person has with an institution, a Web page, part of a Web page, an entire collection of pages or a Web site. In the context of RDF, a resource is *uniquely identified* by a URI (Universal Resource Identifier) [40].

- The building block of the RDF data model is a *triple*. Any RDF expression is a *collection* of triples. A triple is of the form (*subject*, *predicate*, *object*) where *predicate* (also called *property*) denotes a relationship between *subject* and *object*. The subject and predicate of a statement are URIs and the object can be a URI or a *literal*.

A collection of RDF triples can be viewed as a *directed labeled graph* with subjects and objects being the *nodes* of the graph and predicates the *edges*.

Although the RDF Data Model provides a simple way to describe resources and their interrelationships, it doesn't allow the definition of *types (classes)* of resources and the declaration of properties between resources. This is the role of *RDF Schema (RDFS)* [6] which is RDF's vocabulary description language and is considered as a semantic extension of RDF. It provides mechanisms for describing *classes* (i.e., groups of related resources) and *properties* (i.e., relationships between classes). The RDFS language consists of a collection of RDF resources that can be used to describe properties of other RDF resources in application-specific RDF vocabularies. We are going to focus here on the building blocks of RDF Schema Language and provide our view of their semantics : *Classes* and *Properties*.

Classes: Resources may be divided into groups called *classes*. The members of a class are known as *instances* of the class. Classes are themselves resources (recall that in RDF everything is a resource). They are identified by RDF URI References and may be described using RDF properties. We can state that a resource is an instance of a class using the type property that will be discussed later.

RDF distinguishes between a *class* and the *set of its instances*. Associated with each class is a set, called the *class extension* of the class, which is the set of the instances of the class. Two classes may have the same set of instances but be different classes.

RDFS allows one to define that a class C is a *subclass* (*à* la Object Oriented Paradigm) of a class C'. When a class C is a *subclass* of a class C', all instances of C are also instances of C'. The sc property that we will be discussed later is used to state that one class is a subclass of another. The term *superclass* is used as the inverse of subclass.

Some of the built-in class types defined by the RDFS specification are:

- class is the class of resources that are RDF classes. class is an instance of itself.

  res**:** Everything described in RDF is called a resource and is an instance of the class res. This is the class of all resources. All other classes are subclasses of this class. res is an instance of class.

- lit is the class of literal values such as strings and integers. Property values such as textual strings are examples of RDF literals. lit is an instance of class and a subclass of res.

- prop is the class of RDF properties. prop is an instance of class.

Properties: In RDF, a property is a *relationship* between a subject and an object. In contrast to Object Oriented data models properties can also be organized into subsumption hierarchies. If a property P is a *subproperty* of property P', then all pairs of resources which are related by P are also related by P'. The subproperty relation between two properties P and P' is defined with the RDFS property sp. The term *superproperty* is often used as the *inverse* of subproperty.

Some basic built-in properties of the RDFS specification are as follows:

- domain is an RDFS property, instance of prop that is used to state that any resource that has a given property is an instance of one or more classes.
- range is an instance of prop that is used to state that the values of a property are instances of one or more classes.
- type is an instance of prop that is used to state that a resource is an instance of a class.
- sc is an instance of prop that is used to state that all the instances of one class are instances of another. The relation sc is reflexive and transitive.
- sp is an instance of prop that is used to state that all resources related by one property are also related by another. The specification does not define a top property, i.e., a property that is a super-property of all properties. The relation sp is reflexive and transitive.

There are a number of other class and properties defined by the RDFS specification and can be found in [6].

In our framework resources are classified into layers, namely *data*, *schema* and *metaschema* by taking into account the abstraction "level" to which they belong, i.e., their ability to contain members [20]. In the lower abstraction layer, the *data layer* we have the descriptions of *individual* information resources. At the *schema layer* we have classes, which represent abstract entities referring collectively to sets of similar individuals, and properties, which represent attributes or relationships among classes. The upper abstraction layer (*metaschema layer* or RDFS layer) is used to group schema entities into semantic units called *metaclasses*. In our formalisation of RDF and RDFS we also introduce some specific intuitive constraints. Primarily, we do not allow cycles in subclass and subproperty relationships and restrict every property to have one and only one domain and range. We will discuss the above notions in Section 3.2.4 in detail.

3.2 Formalizing RDF

In this section we are going to provide a formal description of RDF and RDFS to set the basis for our work. We are first going to define the notions of RDF *triple* and RDF *graph* and then we are going to discuss *inference*, *typing* and *validity* of RDF graphs.

3.2.1 RDF Graphs

We consider two disjoint, infinite and non-empty sets $\mathbf{U}$, $\mathbf{L}$, denoting the URIs and literals respectively.

Definition 1 *An RDF triple $t = (subject, property, object)$ is any element of the set $\mathbf{U} \times \mathbf{U} \times (\mathbf{U} \cup \mathbf{L})$. We denote by $\mathcal{T}$ the set of triples, i.e. $\mathcal{T} = \mathbf{U} \times \mathbf{U} \times (\mathbf{U} \cup \mathbf{L})$.*

It should be stressed that in this work, we are interested only in *ground triples* and thus ignore non-universally identified resources, called *unnamed* or *blank nodes* [14]. From this point on, and w.l.g. we will be using the term triples to refer to ground triples. In this context, an *RDF graph*[1] is formed by a set of RDF (ground) triples.

Definition 2 *An RDF graph g_r is a set of triples ($g_r \subseteq \mathcal{T}$). We denote by $\mathcal{R}$ the set of RDF Graphs.*

3.2.2 RDFS Graphs

A vocabulary of *property types* but also of *classes* employed to describe resources can be defined in the RDF Schema (RDFS) language [6]. An RDFS schema vocabulary (called namespace [5]) is also represented in the core RDF model under the form of triples and provides (a) abstraction mechanisms, such as (multiple) class or property subsumption and (multiple) classification of resources; (b) specification of domain and range classes to which properties can apply; (c) documentation facilities for names defined in a schema.

The most important RDFS classes are (in brackets the name we will use in this paper) *Resource* [res], *Class* [class] and *Property* [prop]. To assert that a resource is an instance of a class, the RDFS built-in property rdf:type [type] is used. One can assert *subsumption* relationships of classes and properties types with the RDFS subclassOf [sc] and subpropertyOf [sp] built-in properties. Furthermore, RDFS domain [domain] and range [range] properties allow one to specify domain and range classes to which properties can apply.

[1] Formally speaking, RDF graphs are not classical *directed labeled graphs*, because the edge and node labels in an RDF graph are intermingled in the sense that edge names are often used as nodes. Thus, a resource (e.g., *associatedWith*) may occur both as a predicate in ($\&dopamine_receptor_D2$, $associatedWith$, $\&schizophrenia$) and as a subject in ($associatedWith$, domain, $Receptor$) triples. Moreover, an RDFS predicate (e.g., subpropertyOf) may relate other predicates (e.g., $bindsTo$ and $bindsToNeuroTrans$). Thus, the resulting structure is not a graph in the strict mathematical sense.

3.2.3 RDFS Inference

The RDFS specification [15] relies on a set of *inference rules* which, when applied to an RDF graph, entail knowledge which was not explicitly specified. In the sequel, we are referring to inferred triples obtained by applying RDFS inference rules on a set of *explicit* triples as *implicit*. In our work, both explicit and implicit triples are considered to be of equal value.

Reflexivity of sc	$I^{(1)}$:	$\frac{(C,\ \mathsf{type},\ \mathsf{class})}{(C,\ \mathsf{sc},\ C)}$
Transitivity of sc	$I^{(2)}$:	$\frac{(C_1,\ \mathsf{sc},\ C_2),(C_2,\ \mathsf{sc},\ C_3)}{(C_1,\ \mathsf{sc},\ C_3)}$
Reflexivity of sp	$I^{(3)}$:	$\frac{(P,\ \mathsf{type},\ \mathsf{prop})}{(P,\ \mathsf{sp},\ P)}$
Transitivity of sp	$I^{(4)}$:	$\frac{(P_1,\ \mathsf{sp},\ P_2),(P_2,\ \mathsf{sp},\ P_3)}{(P_1,\ \mathsf{sp},\ P_3)}$
Transitivity of class instantiation	$I^{(5)}$:	$\frac{(x,\ \mathsf{type},\ C_1),(C_1,\ \mathsf{sc},\ C_2)}{(x,\ \mathsf{type},\ C_2)}$
Transitivity of property instantiation	$I^{(6)}$:	$\frac{(P_1,\ \mathsf{sp},\ P_2),(x_1,\ P_1,\ x_2)}{(x_1,\ P_2,\ x_2)}$

Table 3.1: Part of RDFS Inference Rules

Given an RDF graph g_r, its implicit triples can be determined by iteratively applying the inference rules shown in Table 3.1, adding triples in the process, until no further rules can be applied (in the style of [14]). The rules in Table 3.1 capture the *reflexivity* and *transitivity* properties of sc and sp as well as the *transitivity* of class and property instantiations. The *closure* of an RDF graph as well as the employed inference mechanism are as usual abstracted by a *consequence operator*, Cn. More formally, for an RDF graph g_r the result of $Cn(g_r)$ contains *all the implicit and explicit* triples of the RDF graph g_r. Using Cn, one can easily define an *implication relation* between RDF graphs:

Definition 3 *An RDF graph $g_r^{(1)}$ implies an RDF graph $g_r^{(2)}$ (denoted by $g_r^{(1)} \vdash g_r^{(2)}$) iff $Cn(g_r^{(2)}) \subseteq Cn(g_r^{(1)})$.*

3.2.4 Typing

One of the critical problems while reading RDF documents is, given a URI (say u) that is used in certain triples in the document, to be able to understand what *type* of object (e.g., individual, class, property, metaclass etc) this URI corresponds to. Typing is critical in the context of programming languages as well as in databases (for optimization, storage

and querying purposes). In our context, typing is central because knowing the types of the various objects is heavily used as part of the inference and validation mechanisms in our platform.

The formalities involved in being able to associate types to objects represented by URIs have been described in [20] and will not be repeated here. The general idea is that, each of the received triples provides some information on the type of (some of) the involved URIs, which eventually allows us to attribute a type to any RDF resource. As an example, the triple (A, sc, class) provides the information that A is a metaclass; given this information, we can infer that B, appearing in the triple (B, type, A) is a schema class.

In general, following the idea from [20], we have identified five types, namely: Individual, SchemaClass, MetaClass, Propertyand MetaProperty. A basic assumption in our context is that an object (URI) cannot be an instance of multiple types, i.e., an object can be either an Individual, a SchemaClass, a MetaClass, a Property, or a MetaProperty, but cannot be any of those two at the same time. A short description of the properties of each of the five types follows:

- An *individual*, i.e., a URI of type Individual, is a URI denoting a specific information source. Individuals belong to the data layer, and are used to denote specific objects. They are instances of classes at the schema layer.

- A *schema class*, i.e., a URI of type SchemaClass, is a URI denoting a collection of individuals. Schema classes belong to the schema layer and they are instances of classes at the metaschema layer. Instances of schema classes are individuals, which belong to the data layer. Also note that schema classes is not the only type of "class" that exists in our typing system; in particular, the term *class* will refer to a URI that is either a schema class, or a metaclass, or a metaproperty.

- A *metaclass*, i.e., a URI of type MetaClass, is a URI denoting a collection of schema classes (For example *RealWorldObject*). In effect, metaclasses are classes which contain other classes. Metaclasses belong to the metaschema layer and their instances are schema classes.

- A *property*, i.e., a URI of type Property, is a URI denoting an attribute of some class (schema class, metaclass or metaproperty). Properties may belong to any of the three layers, depending on their domain and range. They are used to denote relationships between objects. The instances of properties are usually pairs of individuals (but can sometimes be pairs of other objects as well, depending on the layer that the property belongs to). The instances of properties will be called *property instances*.

- A *metaproperty*, i.e., a URI of type MetaProperty, is a URI denoting a collection of schema properties. Since metaproperties are collections of other objects, they are actually classes, but they are different from schema classes or metaclasses in the sense that their instances are schema properties. Metaproperties belong to the metaschema layer.

The above properties apply to user defined URIs. URIs defined in the RDF/S often deviate from the above and have special properties which will prove crucial in the definition of our typing system; in particular:

1. class is a metaclass. All metaclasses should be (explicit or implicit) subclasses of class, i.e., class is the root of all metaclasses. All schema classes should be (explicit or implicit) instances of class. The triple (A, type, class) should exist for all schema classes, metaclasses and metaproperties A, including class itself.

2. res is a schema class. All schema classes should be (explicit or implicit) subclasses of res, i.e., res is the root of all schema classes. All individuals should be (explicit or implicit) instances of res. res itself is an instance of class.

3. propis a metaproperty. All metaproperties should be (explicit or implicit) subclasses of prop, i.e., prop is the root of all metaproperties. All properties should be (explicit or implicit) instances of prop. prop itself is an instance of class.

4. sp, sc, type, domain, range are all properties at the metaschema level. The domain and range of sp is prop.

Following the RDF Schema specification document (and the proposal from [20]), and in order to determine the types of RDF resources, we provide three sets of typing rules:

- set *RDFS* (see Table 3.2) consists of typing rules that allow one to attribute one of our data types to one of the RDFS constructs (class, res, etc.);

- set O (see Table 3.3) consists of typing rules that allow one to associate an RDF resource specified in a user ontology to one of the RDFS data types, usually used when the type of either the subject or the object of the triple is known; and

- set D (see Table 3.4) consists of typing rules specifying some "default" behavior to be used when the other typing rules don't allow a type inference to be made for some RDF resource.

$RDFS_1$: $\overline{\text{class } is\ of\ type \text{ MetaClass}}$

$RDFS_2$: $\overline{\text{prop } is\ of\ type \text{ MetaProperty}}$

$RDFS_3$: $\overline{\text{res } is\ of\ type \text{ SchemaClass}}$

$RDFS_4$: $\overline{\text{domain } is\ of\ type \text{ MetaProperty}}$

$RDFS_5$: $\overline{\text{range } is\ of\ type \text{ MetaProperty}}$

$RDFS_6$: $\overline{\text{sp } is\ of\ type \text{ MetaProperty}}$

$RDFS_7$: $\overline{\text{sc } is\ of\ type \text{ MetaProperty}}$

$RDFS_8$: $\overline{\text{type } is\ of\ type \text{ MetaProperty}}$

Table 3.2: Basic Inference Rules

Note that in Tables 3.3 and 3.4, the URIs denoted by A, C, D are user defined resources, whereas B can be either a user defined resource or one of class, res, prop. Moreover, the triples indicated in the rules may be there either explicit or implicit. The rules (in all tables) are assumed to be taken over the entire set of triples constituting an RDF KB, i.e., the order does not count. As mentioned, the set of rules in Table 3.4 provide a "default" behavior, when the types of the subject and object of the triple under question is unknown. Note that the term "unknown" in this respect refers to the types being unknown *after* the entire set of triples has been read, not at some intermediate point.

3.2.5 Validity

In our work, we consider RDF graphs supporting a notion of validity, which overrules certain triple combinations (in the style of [36] as well as other fragments of SW languages [8, 24]).

We will discriminate between two types of validity constraints, namely *conflict-free* and *completeness*. A *conflict-free* set of triples is one that does not contain contradictory information: for instance, a class cannot be a property at the same time (i.e. disjointness of **C** and **P**). A set of triples is *complete* if it contains all the necessary triples to guarantee validity i.e., there are no *missing triples*. A more formal way to view the difference between a conflict-free and a complete set of triples is by saying that a completeness invalidity can be resolved by the addition of one or more triples, whereas a conflict-free invalidity cannot be resolved by the addition of any set of triples (only by the deletion of one or more). For instance, the completeness invalidity of a property with no domain (see rule CV_3 in

$$O_1: \quad \frac{A \text{ sc } B \textit{ and } (B \textit{ is of type } \text{MetaClass} \textit{ or } A \textit{ is of type } \text{MetaClass})}{A \textit{ is of type } \text{MetaClass} \textit{ and } B \textit{ is of type } \text{MetaClass}}$$

$$O_2: \quad \frac{A \text{ sc } B \textit{ and } (B \textit{ is of type } \text{MetaProperty} \textit{ or } A \textit{ is of type } \text{MetaProperty})}{A \textit{ is of type } \text{MetaProperty} \textit{ and } B \textit{ is of type } \text{MetaProperty}}$$

$$O_3: \quad \frac{A \text{ sc } B \textit{ and } (B \textit{ is of type } \text{SchemaClass} \textit{ or } A \textit{ is of type } \text{SchemaClass})}{A \textit{ is of type } \text{SchemaClass} \textit{ and } B \textit{ is of type } \text{SchemaClass}}$$

$$O_4: \quad \frac{A \text{ sp } B}{A \textit{ is of type } \text{Property} \textit{ and } B \textit{ is of type } \text{Property}}$$

$$O_5: \quad \frac{A \text{ type } B \textit{ and } B \textit{ is of type } \text{MetaClass} \textit{ and } B \neq \text{class}}{A \textit{ is of type } \text{SchemaClass}}$$

$$O_6: \quad \frac{A \text{ type } B \textit{ and } B \textit{ is of type } \text{MetaProperty}}{A \textit{ is of type } \text{Property}}$$

$$O_7: \quad \frac{A \text{ type } B \textit{ and } B \textit{ is of type } \text{SchemaClass}}{A \textit{ is of type } \text{Individual}}$$

$$O_8: \quad \frac{A \text{ type } B \textit{ and } A \textit{ is of type } \text{SchemaClass}}{A \textit{ is of type } \text{MetaClass}}$$

$$O_9: \quad \frac{A \text{ type } B \textit{ and } A \textit{ is of type } \text{Property}}{A \textit{ is of type } \text{MetaProperty}}$$

$$O_{10}: \quad \frac{A \text{ type } B \textit{ and } A \textit{ is of type } \text{Individual}}{A \textit{ is of type } \text{SchemaClass}}$$

$$O_{11}: \quad \frac{A \text{ domain } B}{A \textit{ is of type } \text{Property}}$$

$$O_{12}: \quad \frac{A \text{ range } B}{A \textit{ is of type } \text{Property}}$$

$$O_{13}: \quad \frac{A\ C\ D}{C \textit{ is of type } \text{Property}}$$

$$O_{14}: \quad \frac{A\ C\ D \textit{ and } C \text{ domain } B \textit{ and } B \textit{ is of type } \text{SchemaClass}}{A \textit{ is of type } \text{Individual}}$$

$$O_{15}: \quad \frac{A\ C\ D \textit{ and } C \text{ domain } B \textit{ and } B \textit{ is of type } \text{MetaClass}}{A \textit{ is of type } \text{SchemaClass}}$$

$$O_{16}: \quad \frac{A\ C\ D \textit{ and } C \text{ range } B \textit{ and } B \textit{ is of type } \text{SchemaClass}}{D \textit{ is of type } \text{Individual}}$$

$$O_{17}: \quad \frac{A\ C\ D \textit{ and } C \text{ range } B \textit{ and } B \textit{ is of type } \text{MetaClass}}{D \textit{ is of type } \text{SchemaClass}}$$

Table 3.3: Typing Rules

Table 3.6) can be resolved by adding a domain to the property (as well as by removing the property itself). On the other hand, a cyclic sc (see rule CFV_2 in Table 3.5) will not cease

$$D_1: \frac{A \text{ type class } and\ not(A\ is \text{ MetaClass } or\ A\ is \text{ MetaProperty })}{A\ is \text{ SchemaClass}}$$

$$D_2: \frac{A \text{ type } B\ and\ not(A\ is \text{ SchemaClass } orA\ is \text{ Property } orB\ is \text{ MetaClass } orB\ is \text{ MetaProperty})}{A\ is \text{ Individual } and\ B\ is \text{ SchemaClass}}$$

$$D_3: \frac{A \text{ sc } B\ and\ not(A\ is \text{ MetaClass } orA\ is \text{ MetaProperty } orB\ is \text{ MetaClass } orB\ is \text{ MetaProperty})}{A\ is \text{ SchemaClass } and\ B\ is \text{ SchemaClass}}$$

$$D_4: \frac{A \text{ domain } B\ and\ not(B\ is \text{ MetaClass } or\ B\ is \text{ MetaProperty })}{B\ is \text{ SchemaClass}}$$

$$D_5: \frac{A \text{ range } B\ and\ not(B\ is \text{ MetaClass } or\ B\ is \text{ MetaProperty })}{B\ is \text{ SchemaClass}}$$

$$D_6: \frac{A\ C\ D\ and\ not\ A\ is \text{ SchemaClass}}{A\ is \text{ Individual}}$$

$$D_7: \frac{A\ C\ D\ and\ not\ D\ is \text{ SchemaClass}}{D\ is \text{ Individual}}$$

Table 3.4: Default Typing Rules

Disjoint classes & properties	CFV_1:	$\frac{\forall u \in \mathbf{U}, u\ is\ of\ type\ x, u\ is\ of\ type\ y}{x = y}$
Acyclic sc	CFV_2:	$\frac{\forall x, y \in \mathbf{C}, (x,\ \text{sc},\ y), (y,\ \text{sc},\ x)}{x \neq y}$
Acyclic sp	CFV_3:	$\frac{\forall x, y \in \mathbf{P}, (x,\ \text{sp},\ y), (y,\ \text{sp},\ x)}{x \neq y}$
Unique property domain	CFV_4:	$\frac{\forall p \in \mathbf{P}, u, v \in \mathbf{C} (p,\ \text{domain},\ u), (p,\ \text{domain},\ v)}{u = v}$
Unique property range	CFV_5:	$\frac{\forall p \in \mathbf{P}, u, v \in \mathbf{C} (p,\ \text{range},\ u), (p,\ \text{range},\ v)}{u = v}$

Table 3.5: Conflict Free Validity Rules

to exist no matter what triples we add; our only choice is to remove some sc relationship to break the cycle, so this is a conflict-free invalidity.

Using this distinction, we can now state our requirement regarding validity as follows: the set of triples for a given RDF/S graph should be conflict free. Tables 3.5 and 3.6 illustrate the rules that ensure conflict-free and completeness validity respectively. A rule

$\frac{R}{R'}$ in these tables should be read as follows: if R holds in a set of triples S, then R' should hold for S to be conflict-free/complete.

sc constraint for property domain	CV_1 :	$\dfrac{p, q \in \mathbf{P}, u, v \in \mathbf{C}, (p,\ \mathsf{domain},\ u), (q,\ \mathsf{domain},\ v), (p,\ \mathsf{sp},\ q)}{(u,\ \mathsf{sc},\ v)}$
sc constraint for property range	CV_2 :	$\dfrac{p, q \in \mathbf{P}, u, v \in \mathbf{C}, (p,\ \mathsf{range},\ u), (q,\ \mathsf{range},\ v), (p,\ \mathsf{sp},\ q)}{(u,\ \mathsf{sc},\ v)}$
type constraint for property domain	CV_3 :	$\dfrac{p \in \mathbf{P}, c \in \mathbf{C}, u, v \in \mathbf{U}, (p,\ \mathsf{domain},\ c), (u,\ p,\ v)}{(u,\ \mathsf{type},\ c)}$
type constraint for property range	CV_4 :	$\dfrac{p \in \mathbf{P}, c \in \mathbf{C}, u, v \in \mathbf{U}, (p,\ \mathsf{range},\ c), (u,\ p,\ v)}{(v,\ \mathsf{type},\ c)}$

Table 3.6: Completeness Validity Rules

Definition 4 *A set of S of triples $(s,\ p,\ o)i$ is valid iff it is both conflict-free and complete.*

Chapter 4

Named Graphs and Graphsets

Named graphs were introduced as a solution to the RDF graph modularization problem. They can be seen as an extension of the RDF *reification* mechanism [12], which allows one to objectify (i.e. assign a URI to) a single statement and use it as resource in other triples. Named graphs allow us to objectify a *set* of triples and assert other statements (e.g., regarding provenance) about this set.

It is worth noticing, that the modularity offered by *namespaces* [5] does not offer the same functionality. First, namespaces were introduced as a means to prefix class and property names, rather than modularize the triples themselves. Secondly, unlike namespaces which are by definition mutually disjoint, named graphs may share the same triples.

4.1 Named Graphs

Intuitively, an RDF named graph is an RDF graph (or part of it) to which we have assigned a URI. Recall that both contain triples, rather than resources; a node (e.g. a URI denoting a resource) does not itself belong to any named graph, even though the triples referring to the node may belong to one or more named graphs. Let $\mathbf{G} \subset \mathbf{U}$ be the set of named graph URIs. Formally:

Definition 5 *A named graph g_n is a pair $g_n = (n, \gamma(n))$, where $n \in \mathbf{G}$ and γ is a function $\gamma : \mathbf{G} \mapsto \mathcal{R}$. We will denote by $\mathcal{G}$ the set of all named graphs.*

Function γ associates names from $\mathbf{G}$ with sets of triples i.e., RDF Graphs from $\mathcal{R}$. Unlike [9], we view an RDF named graph as a *function* from the set of named graph URIs to the set of RDF graphs. Thus, our definition dictates that a name (URI) n can only be associated with *exactly one* RDF graph. In this setting, we distinguish between the

URIs used to name RDF named graphs ($\mathbf{G} \subset \mathbf{U}$) and the URIs of classes ($\mathbf{C} \subset \mathbf{U}$) and properties ($\mathbf{P} \subset \mathbf{U}$).

Definition 6 *For every named graph $g_n = (n, g_r) \in \mathcal{G}$ we define the following functions:*

- $\texttt{name} : \mathcal{G} \mapsto \mathbf{G}$ *where* $\texttt{name}(g_n) = n$
- $\texttt{rdfgraph} : \mathcal{G} \mapsto \mathcal{R}$ *where* $\texttt{rdfgraph}(g_n) = g_r = \gamma(n)$
- $\texttt{closure} : \mathcal{G} \mapsto \mathcal{R}$ *where* $\texttt{closure}(g_n) = Cn(g_r)$

Functions `name`, `rdfgraph` and `closure` allow us to access the internals of a named graph. More specifically, for a named graph $g_n = (n, g_r)$, function `name` returns its name (URI), and functions `rdfgraph` and `closure` return g_r and $Cn(g_r)$ respectively where Cn is the consequence operator discussed in Section 3.2.3.

Example 1. *For the named graph $g_n^{(1)} = (n^{(1)}, g_r^{(1)})$ with $g_r^{(1)} = \{(x,\ \texttt{type},\ A), (A,\ \texttt{sc},\ B)\}$:*

$$\begin{aligned} \texttt{name}(g_n^{(1)}) &= n^{(1)} \\ \texttt{rdfgraph}(g_n^{(1)}) &= \{(x,\ \texttt{type},\ A), (A,\ \texttt{sc},\ B)\} \\ \texttt{closure}(g_n^{(1)}) &= \{(x,\ \texttt{type},\ A), (A,\ \texttt{sc},\ B), (x,\ \texttt{type},\ B)\} \end{aligned}$$

In addition to the above functions, we define a set of relations for named graphs. Let $g_n^{(1)}$ and $g_n^{(2)}$ be two named graphs.

We say that $g_n^{(1)}$ and $g_n^{(2)}$ are *equal*, denoted by $g_n^{(1)} \doteq g_n^{(2)}$ iff they have the same name:

$$g_n^{(1)} \doteq g_n^{(2)} \Leftrightarrow \texttt{name}(g_n^{(1)}) = \texttt{name}(g_n^{(2)})$$

Two equal named graphs are considered *identical*.

We say that $g_n^{(1)}$ *implies* $g_n^{(2)}$, denoted by $g_n^{(1)} \vdash g_n^{(2)}$ iff the closure of $g_n^{(2)}$ is a subset of the closure of $g_n^{(1)}$:

$$g_n^{(1)} \vdash g_n^{(2)} \Leftrightarrow \texttt{closure}(g_n^{(2)}) \subseteq \texttt{closure}(g_n^{(1)})$$

Note that the symbol $\vdash$ was also used for the implication relation between RDF graphs.

Example 2. *Consider named graph $g_n^{(1)}$ from Example 1 and $g_n^{(2)} = (n^{(2)}, g_r^{(2)})$ with $g_r^{(2)} = \{(x,\ \texttt{type},\ B)\}$. Then: $g_n^{(1)} \vdash g_n^{(2)}$.*

Moreover, we define the *union* operation between named graphs $g_n^{(1)}$ and $g_n^{(2)}$, denoted by $g_n^{(1)} \odot g_n^{(2)}$, to be an RDF graph (i.e., set of triples) defined as follows:

$$g_n^{(1)} \odot g_n^{(2)} = \texttt{rdfgraph}(g_n^{(1)}) \cup \texttt{rdfgraph}(g_n^{(2)})$$

Finally, we say that triple $(s,\ p,\ o)$ *belongs to* named graph $g_n^{(1)}$ iff it belongs to its closure. For readability reasons we write $(s, p, o, n^{(1)})$ to denote this fact. Formally, for $n^{(1)} = \texttt{name}(g_n^{(1)})$:

$$(s, p, o, n^{(1)}) \Leftrightarrow \{(s,\ p,\ o)\} \in \texttt{closure}(g_n^{(1)})$$

Example 3. *For the named graph $g_n^{(1)}$ of Example 1 and triple $(x,\ \texttt{type},\ A)$ we write $(x, \texttt{type}, A, n^{(1)})$.*

4.2 RDF/S GraphSets

As previously discussed, RDF/S *graphsets* contain all the triples that are *jointly owned* by a set of named graphs, but are not owned by any subset thereof. To uniquely identify RDF/S *graphsets* we consider a set $\mathcal{I} \subset \mathbf{U}$ which includes the set of named graph URIs, $\mathbf{G}$ (i.e. $\mathbf{G} \subset \mathcal{I}$) and is pairwise disjoint with sets $\mathbf{C}$ and $\mathbf{P}$ of class and property names respectively. Then:

Definition 7 *A graphset is a pair of the form*

$$g_s = (\texttt{sid}(N), \Gamma(\texttt{sid}(N)))$$

where:

- $N \subseteq \mathbf{G}$ *is the set of RDF named graphs' names that comprise the graphset*
- `sid` *is a skolem function that assigns an id to a set of RDF named graph names* $(sid : 2^{\mathbf{G}} \mapsto \mathcal{I})$ *with* $\texttt{sid}(\{n\}) = n$ *for all* $n \in \mathbf{G}$
- Γ *is a function* $\Gamma : \mathcal{I} \mapsto \mathcal{R}$

We denote with $\mathcal{GS}$ the set of all graphsets.

The intuition behind function Γ is the same as for function γ for named graphs: it returns the *explicit* triples that are associated with the graphset *directly* and not through its constituent named graphs. Note that this feature is essential to record and reason about RDF/S graph provenance information and it is not supported by other generalizations of named graphs such as those proposed in [34]. In this work, authors define named graphs

intentionally through SPARQL views. In the same spirit as with named graphs, we define functions that allow us to access the ingredients of a graphset:

Definition 8 *For every graphset* $g_s = (\texttt{sid}(N), \Gamma(\texttt{sid}(N))) \in \mathcal{GS}$ *we define the following functions:*

- $\texttt{gid} : \mathcal{GS} \mapsto \mathcal{I}$ *where* $\texttt{gid}(g_s) = \texttt{sid}(N)$*;*
- $\texttt{ngnms} : \mathcal{GS} \mapsto 2^{\mathbf{G}}$ *where* $\texttt{ngnms}(g_s) = N$*;*
- $\texttt{ngs} : \mathcal{GS} \mapsto 2^{\mathcal{G}}$ *where* $\texttt{ngs}(g_s) = \{g_n \in \mathcal{G} : \texttt{name}(g_n) \in N\}$*;*
- $\texttt{rdfgraph} : \mathcal{GS} \mapsto \mathcal{R}$ *where* $\texttt{rdfgraph}(g_s) = \Gamma(\texttt{sid}(N))$*;*
- $\texttt{closure} : \mathcal{GS} \mapsto \mathcal{R}$ *where*

$$\begin{aligned}\texttt{closure}(g_s) = Cn\Big(\quad & \Gamma(\texttt{sid}(N)) \bigcup \{t \mid \forall g_n^{(i)} \in \texttt{ngs}(g_s), \\ & \exists t_{ij} \in \texttt{closure}(g_n^{(i)}), j = 1, 2, \ldots, m_i, m_i \geq 1 \\ & s.t.\ \textstyle\bigcup_{i=1,\ldots,k} \bigcup_{j=1,\ldots,m_i} \{t_{ij}\} \vdash \{t\}\}\Big)\end{aligned}$$

Functions `gid` and `ngnms` return the URIs of the graphset and its constituent named graphs respectively, while `ngs` returns the named graphs themselves. Functions `rdfgraph` and `closure` return the set of triples of the graphset and its closure respectively.

The functions' definitions are straightforward, with the exception of `closure`. Consider the set S of named graphs $g_n^{(i)} = (n^{(i)}, g_r^{(i)}), i = 1, \ldots, k$ and the graphset $g_s = (\texttt{sid}(N), \Gamma(\texttt{sid}(N)))$, where $N = \{n^{(1)}, \ldots, n^{(k)}\}$.
The `closure` function is not $Cn(\Gamma(\texttt{sid}(N)))$ as for named graphs; such a definition would only capture the triples explicitly asserted in the graphset via the function Γ. Here, we also need to capture the triples *jointly owned* by the named graphs $g_n^{(i)}$ that form the graphset; these triples are the ones *implied* by the triples in $g_r^{(1)} \odot \ldots \odot g_r^{(k)}$ but are *not contained* in any graphset that is constituted by a subset S' of S (see Definition 8).

Example 4. *Consider the named graphs*

1. $g_n^{(1)} = (n^{(1)}, \{(r, \texttt{type}, A), (A, \texttt{sc}, B), (C, \texttt{sc}, D)\})$,
2. $g_n^{(2)} = (n^{(2)}, \emptyset)$ *and*
3. $g_n^{(3)} = (n^{(3)}, \{(B, \texttt{sc}, C)\})$;

shown in Fig. 4.2(a), and graphset $g_s = (\texttt{sid}(\{n^{(1)}, n^{(2)}, n^{(3)}\}), g_r)$ *with* $g_r = \{(x, \texttt{type}, D)\}$. *Then,* $\texttt{closure}(g_s) = g_r \cup \emptyset$ *since there does not exist a triple* t *that is inferred by triples belonging in all of the named graphs* $g_n^{(1)}, g_n^{(2)} g_n^{(3)}$.

Suppose that named graph $g_n^{(2)}$ *is now* $g_n^{(2)} = (n^{(2)}, \{(A,\ \mathsf{sc},\ B)\})$ *(see Fig. 4.2(b)). Then,*

$$\texttt{closure}(g_s) = g_r \cup \{(r,\ \mathsf{type},\ C), (A,\ \mathsf{sc},\ D)\}$$

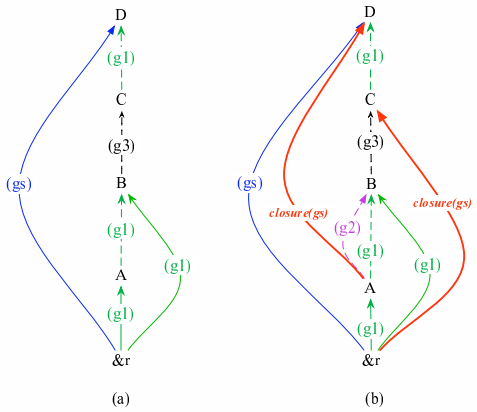


Figure 4.1: Graphset Example

It should be stressed that a named graph $g_n = (n, g_r)$ can be viewed as a *singleton graphset* g_s whose constituent is the named graph g_n (only); thus, $g_s = (\texttt{sid}(\{n\}), g_r)$. Given that, per Definition 7, $\texttt{sid}(\{n\}) = n$, the graphset becomes $g_s = (n, g_r)$, so g_n is actually a graphset. The application of graphset functions on singleton graphsets is straightforward: $\texttt{gid}(g_s) = n$, $\texttt{ngnms}(g_s) = \{n\}$, $\texttt{ngs}(g_s) = g_n$, $\texttt{rdfgraph}(g_s) = g_r$, $\texttt{closure}(g_s) = Cn(g_r)$. From this point on and w.l.g. we will be using the term graphsets to refer to named graphs as well. Furthermore, graphsets *can be materialized* and subsequently treated as RDF named graphs by assigning them a user defined URI instead of their internally generated `sid`. The extent of materialized graphsets comprises both explicit and implicit triples and they behave as yet another source of triples: the connection with their constituent named graphs is lost. This is useful for distributed SW applications requiring to exchange graphsets from one RDF/S processing system to another.

In the same spirit as with named graphs, we define a set of relations for graphsets. Let $g_s^{(1)}$ and $g_s^{(2)}$ be two graphsets. We say that $g_s^{(1)}$ and $g_s^{(2)}$ are *equal*, denoted by $g_s^{(1)} \doteqdot g_s^{(2)}$ if and only if they have the same identifier:

$$g_s^{(1)} \doteqdot g_s^{(2)} \Leftrightarrow \texttt{gid}(g_s^{(1)}) = \texttt{gid}(g_s^{(2)})$$

As with named graphs, two equal graphsets are considered *identical*. Note that we are using here the same symbol that we used to denote named graph equality.

We say that a named graph g_n is a *constituent of* graphset $g_s^{(1)}$, denoted by $g_n \prec g_s^{(1)}$ iff g_n belongs to $\texttt{ngs}(g_s^{(1)})$:

$$g_n \prec g_s^{(1)} \Leftrightarrow g_n \in \texttt{ngs}(g_s^{(1)})$$

We say that a triple t *belongs to* graphset $g_s^{(1)}$ iff t belongs to the closure of $g_s^{(1)}$. For readability reasons we write $(s, p, o, i^{(1)})$ to denote this fact. Formally, for $i^{(1)} = \texttt{gid}(g_s^{(1)})$:

$$(s, p, o, i^{(1)}) \Leftrightarrow \{(s,\ p,\ o)\} \in \texttt{closure}(g_s^{(1)})$$

We say that $g_s^{(1)}$ *implies* $g_s^{(2)}$, denoted by $g_s^{(1)} \vdash g_s^{(2)}$ iff the closure of $g_s^{(2)}$ is a subset of the closure of $g_s^{(1)}$:

$$g_s^{(1)} \vdash g_s^{(2)} \Leftrightarrow \texttt{closure}(g_s^{(2)}) \subseteq \texttt{closure}(g_s^{(1)})$$

Recall that the symbol $\vdash$ was also used for the implication relation between RDF graphs and named graphs.

Moreover, given a set S of graphset identifiers (from $\mathcal{I}$), it is useful to identify the graphset whose set of constituent named graphs is the union of the constituent named graphs of the graphsets considered in S. This is the purpose of function $\texttt{gs} : 2^{\mathcal{I}} \rightarrow \mathcal{I}$, where $\texttt{gs}(\{n^{(1)}, \ldots, n^{(k)}\})$ returns the identifier for graphset $g_s = (\texttt{sid}(N), g_r)$ where:

$$N = \bigcup_{\substack{g_s^{(i)}, \texttt{sid}(g_s^{(i)}) = n^{(i)}, \\ i \in \{1, \ldots, k\}}} \texttt{ngnms}(g_s^{(i)})$$

Last but not least, we introduce the notion of *extended RDF/S graphs* featuring triples with graphset information (i.e. quadrables):

Definition 9 *An extended RDF/S graph D is a set of quadruples of the form (s, p, o, i), where there exist a graphset $g_s \in \mathcal{GS}$ with $\texttt{gid}(g_s) = i$ and $(s, p, o) \in \texttt{rdfgraph}(g_s)$.*

4.3 Reasoning on RDF/S Graphsets

In this section we discuss *inference*, *validity* and *redundancy elimination* for extended RDF/S graphs. The validity constraints as well as redundancy elimination (in the style of [36]) are defined *independently* of the notion of graphsets introduced in this paper. We choose to detect inconsistencies and redundancies at update time rather than adding assertions leading to inconsistent and redundant RDF graphs and then checking them at query time (the usual reasoners approach like Racer [30] and Pellet [26]). This choice

is motivated by the concern that in real scale Semantic Web systems query performance should prevail over update performance. Redundant-free RDF graphs offer a number of advantages in the case of transaction management for concurrent updates and queries.

4.3.1 Inference

The RDF/S inference can (and should) be applied across graphsets (including named graphs) as well. The introduction of graphsets causes the additional difficulty that one should record the graphset that the implicit triple belongs to, based on the graphset(s) that the triples implying it already belong to. Therefore, we need to extend our inference rules so that they also take into account graphsets. The result shown in Table 4.1 is a straightforward extension of the RDF/S inference rules depicted in Table 3.1.

As shown in Table 4.1, an implicit triple belongs to the graphset whose constituent named graphs is the union of the constituent named graphs of the graphsets of the triples involved in the implication. Take, for instance, rule $I_g^{(2)}$; if $(A, \mathsf{sc}, B, i^{(1)})$ with $i^{(1)}$ the identifier for graphset $g_s^{(1)}$ whose constituent named graphs are $g_n^{(1)}$ and $g_n^{(2)}$ and $(B, \mathsf{sc}, C, i^{(2)})$ with $i^{(2)}$ the identifier for graphset $g_s^{(2)}$ whose constituent named graphs are $g_n^{(1)}$ and $g_n^{(3)}$, then, by the definition of function $\mathtt{gs}$ (see Section 4.2) and $I_g^{(2)}$ one obtains $(A, \mathsf{sc}, C, i^{(3)})$ with $i^{(3)}$ the identifier of graphset g_s whose constituent named graphs are $g_n^{(1)}$, $g_n^{(2)}$ and $g_n^{(3)}$. Similarly for the other rules. In Table 4.1, $i^{(1)}$ and $i^{(2)}$ are graphset identifiers.

Reflexivity of sc	$I_g^{(1)}$:	$\dfrac{(C, \mathsf{type}, \mathsf{class}, i^{(1)})}{(C, \mathsf{sc}, C, i^{(1)})}$
Transitivity of sc	$I_g^{(2)}$:	$\dfrac{(C_1, \mathsf{sc}, C_2, i^{(1)}), (C_2, \mathsf{sc}, C_3, i^{(2)})}{(C_1, \mathsf{sc}, C_3, \mathtt{gs}(\{i^{(1)}, i^{(2)}\}))}$
Reflexivity of sp	$I_g^{(3)}$:	$\dfrac{(P, \mathsf{type}, \mathsf{prop}, i^{(1)})}{(P, \mathsf{sp}, P, i^{(1)})}$
Transitivity of sp	$I_g^{(4)}$:	$\dfrac{(P_1, \mathsf{sp}, P_2, i^{(1)}), (P_2, \mathsf{sp}, P_3, i^{(2)})}{(P_1, \mathsf{sp}, P_3, \mathtt{gs}(\{i^{(1)}, i^{(2)}\}))}$
Transitivity of class instantiation	$I_g^{(5)}$:	$\dfrac{(x, \mathsf{type}, C_1, i^{(1)}), (C_1, \mathsf{sc}, C_2, i^{(2)})}{(x, \mathsf{type}, C_2, \mathtt{gs}(\{i^{(1)}, i^{(2)}\}))}$
Transitivity of property instantiation	$I_g^{(6)}$:	$\dfrac{(P_1, \mathsf{sp}, P_2, i^{(1)}), (x_1, P_1, x_2, i^{(2)})}{(x_1, P_2, x_2, \mathtt{gs}(\{i^{(1)}, i^{(2)}\}))}$

Table 4.1: Part of RDFS Inference Rules with Graphsets

In a similar manner as with simple RDF graphs the *closure* of an extended RDF/S graph as well as the employed inference mechanism is abstracted by the consequence operator Cn. More formally, for an extended RDF/S graph D, the result of $Cn(D)$

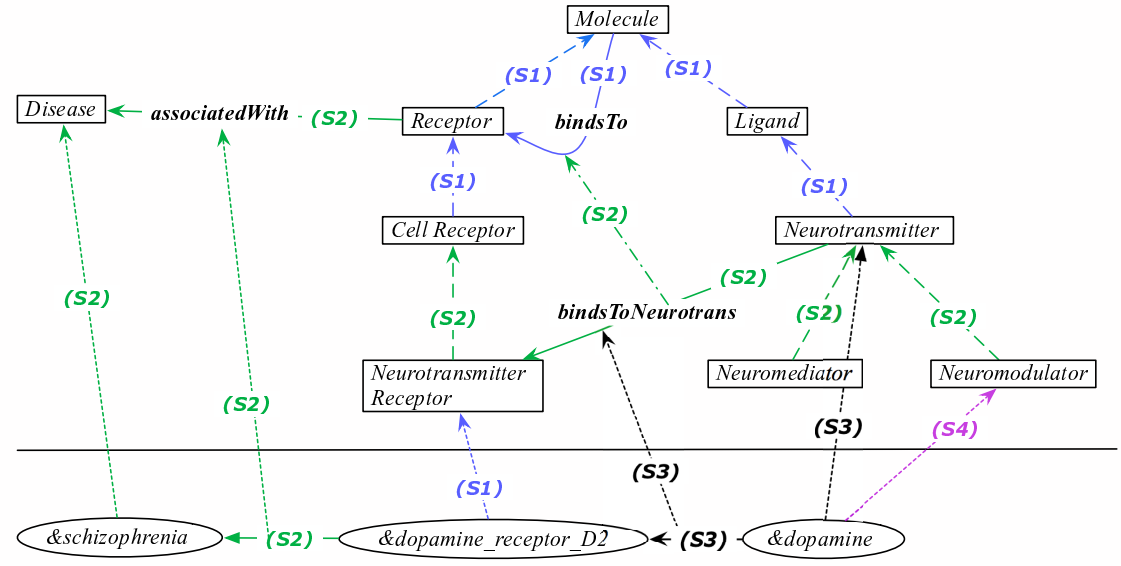


Figure 4.2: Collaborative Neurobiology Ontology

contains all the *explicit* and *implicit* triples where the latter are obtained by applying the inference rules in Table 4.1.

4.3.2 Validity

In our work, we consider RDF graphs supporting a notion of validity, which overrules certain triple combinations (in the style of [36] as well as other fragments of SW languages [8, 24]). However, validity is more complicated in the context of graphsets, as we should determine whether validity should be considered at the level of each graphset, or at the level of an extended RDF/S graph that considers multiple graphsets.

As a motivating example, consider the extended RDF/S graph of Fig. 4.2, where triple (&*dopamine*, *bindsToNeurotrans*, &*dopamine_receptor_D2*) belongs to graphset S_3, but triple (&*dopamine_receptor_D2*, type, *Neurotransmitter Receptor*), which defines that &*dopamine_receptor_D2* is an instance of *Neurotransmitter Receptor*, belongs to graphset S_1. This is a normal situation, so it seems like validity should not be imposed at the level of graphsets, but at the level of the extended RDF/S graph that considers multiple graphsets. On the other hand, there are other types of invalidities concerning a single graphset (e.g., if *Neurotransmitter Receptor* was defined both as a class and as a property) which should be overruled at both the level of each individual graphset (as the graphset would not make sense) and at the level of the extended RDF/S graph that considers the graphset.

This analysis motivated us to discriminate between two types of validity constraints, namely *conflict-free* and *completeness*. A *conflict-free* set of quadruples is one that does

x'Disjoint classes & properties	CFV_1 :	$\frac{\forall u \in \mathbf{U}, u \; is \; of \; type \; x, u \; is \; of \; type \; y}{x = y}$
Acyclic sc	CFV_2 :	$\frac{\forall x, y \in \mathbf{C}, i_1, i_2 \in \mathcal{I}, (x, \mathsf{sc}, y, i_1), (y, \mathsf{sc}, x, i_2)}{x \neq y}$
Acyclic sp	CFV_3 :	$\frac{\forall x, y \in \mathbf{P}, i_1, i_2 \in \mathcal{I}, (x, \mathsf{sp}, y, i_1), (y, \mathsf{sp}, x, i_2)}{x \neq y}$
Unique property domain	CFV_4 :	$\frac{\forall p \in \mathbf{P}, u, v \in \mathbf{C}, i_1, i_2 \in \mathcal{I} (p, \mathsf{domain}, u, i_1), (p, \mathsf{domain}, v, i_2)}{u = v}$
Unique property range	CFV_5 :	$\frac{\forall p \in \mathbf{P}, u, v \in \mathbf{C}, i_1, i_2 \in \mathcal{I} (p, \mathsf{range}, u, i_1), (p, \mathsf{range}, v, i_2)}{u = v}$

Table 4.2: Conflict Free Validity Rules

not contain contradictory information: for instance, a class cannot be a property at the same time (i.e. disjointness of $\mathbf{C}$ and $\mathbf{P}$). A set of quadruples is *complete* if it contains all the necessary triples to guarantee validity i.e., there are no *missing triples*. A more formal way to view the difference between a conflict-free and a complete set of quadruples is by saying that a completeness invalidity can be resolved by the addition of one or more quadruples, whereas a conflict-free invalidity cannot be resolved by the addition of any set of quadruples (only by the deletion of one or more). For instance, the completeness invalidity of a property with no domain (see rule CV_3 in Table 4.3) can be resolved by adding a domain to the property (as well as by removing the property itself). On the other hand, a cyclic sc (see rule CFV_2 in Table 4.2) will not cease to exist no matter what quadruples we add; our only choice is to remove some sc relationship to break the cycle, so this is a conflict-free invalidity.

Using this distinction, we can now state our requirements regarding validity as follows: *i)* the set of quadruples for a given graphset should be conflict free; *ii)* the set of quadruples in an extended RDF/S graph that comprises multiple graphsets should be complete **and** conflict-free. Tables 4.2 and 4.3 illustrate the rules that ensure conflict-free and completeness validity respectively. A rule $\frac{R}{R'}$ in these tables should be read as follows: if R holds in a set of quadruples S, then R' should hold for S to be conflict-free/complete.

Definition 10 *A set of S of quadruples (s, p, o, i) is valid iff it is both conflict-free and complete.*

From the above, it follows that if an extended RDF/S graph is conflict-free then the

sc constraint for property domain	CV_1 :	$\dfrac{p, q \in \mathbf{P}, u, v \in \mathbf{C}, \; i^{(1)}, i^{(2)}, i^{(3)}, i^{(4)} \in \mathcal{I} \; (p, \mathsf{domain}, u, i^{(1)}), (q, \mathsf{domain}, v, i^{(2)}), \; (p, \mathsf{sp}, q, i^{(3)})}{(u, \mathsf{sc}, v, i^{(4)})}$
sc constraint for property range	CV_2 :	$\dfrac{p, q \in \mathbf{P}, u, v \in \mathbf{C}, \; i^{(1)}, i^{(2)}, i^{(3)}, i^{(4)} \in \mathcal{I} \; (p, \mathsf{range}, u, i^{(1)}), (q, \mathsf{range}, v, i^{(2)}), \; (p, \mathsf{sp}, q, i^{(3)})}{(u, \mathsf{sc}, v, i^{(4)})}$
type constraint for property domain	CV_3 :	$\dfrac{p \in \mathbf{P}, c \in \mathbf{C}, u, v \in \mathbf{U}, \; i^{(1)}, i^{(2)}, i^{(3)} \in \mathcal{I} \; (p, \mathsf{domain}, c, i^{(1)}), (u, p, v, i^{(2)})}{(u, \mathsf{type}, c, i^{(3)})}$
type constraint for property range	CV_4 :	$\dfrac{p \in \mathbf{P}, c \in \mathbf{C}, u, v \in \mathbf{U}, \; i^{(1)}, i^{(2)}, i^{(3)} \in \mathcal{I} \; (p, \mathsf{range}, c, i^{(1)}), (u, p, v, i^{(2)})}{(v, \mathsf{type}, c, i^{(3)})}$

Table 4.3: Completeness Validity Rules

graphsets associated with it are conflict-free.

4.3.3 Redundancy Elimination

In an RDF graph, triples that can be inferred through the consequence operator Cn need not be explicitly stored, as they can be obtained from other triples. Such triples are called *redundant*. As in the case of inference and validity, redundancy elimination should take into consideration the presence of graphsets. Intuitively, a quadruple is redundant if there exist quadruples that infer it per the inference rules of Table 4.1. Formally:

Definition 11 *Let S be a set of quadruples. Quadruple (s, p, o, i) is redundant in S iff there exist quadruples $(s_j, p_j, o_j, i^{(j)}) \in S, j = 1, \ldots, k$ such that $\{(s_j, p_j, o_j, i^{(j)}) \mid j = 1, \ldots, k\}$ imply (s, p, o, i) per the inference rules of Table 4.1.*

For a set of rules that allow us to perform redundancy elimination, see Table 4.4.

4.3.4 Complexity Analysis

When working with graphsets, there are several problems that we need to deal with, such as determining whether a graphset equals another and whether a graphset implies another. The former is easily addressed; on the other hand, graphset implication can be reduced to the problem of determining whether a triple t *belongs to graphset* g_s or equivalently if

$$RR_1: \quad \frac{(c, \mathsf{type}, \mathsf{class}, i^{(1)})}{(c, \mathsf{sc}, c, i^{(1)}) \ is\ redundant}$$

$$RR_2: \quad \frac{(p, \mathsf{type}, \mathsf{prop}, i^{(1)})}{(p, \mathsf{sp}, p, i^{(1)}) \ is\ redundant}$$

$$RR_3: \quad \frac{(c_1, \mathsf{sc}, c_2, i^{(1)}), (c_2, \mathsf{sc}, c_3, i^{(2)})}{(c_1, \mathsf{sc}, c_3, \mathtt{gs}(\{i^{(1)}, i^{(2)}\})) \ is\ redundant}$$

$$RR_4: \quad \frac{(x, \mathsf{type}, c_1, i^{(1)}), (c_1, \mathsf{sc}, c_2, i^{(2)})}{(x, \mathsf{type}, c_2, \mathtt{gs}(\{i^{(1)}, i^{(2)}\})) \ is\ redundant}$$

$$RR_5: \quad \frac{(p_1, \mathsf{sp}, p_2, i^{(1)}), (p_2, \mathsf{sp}, p_3, i^{(2)})}{(p_1, \mathsf{sp}, p_3, \mathtt{gs}(\{i^{(1)}, i^{(2)}\})) \ is\ redundant}$$

$$RR_6: \quad \frac{(x, \mathsf{type}, p_1, i^{(1)}), (p_1, \mathsf{sp}, p_2, i^{(2)})}{(x_1, \mathsf{type}, p_2, \mathtt{gs}(\{i^{(1)}, i^{(2)}\})) \ is\ redundant}$$

Table 4.4: Redundancy Elimination Rules

(s, p, o, i), with i the identifier of graphset g_s. Recall that a triple t belongs to graphset g_s iff $t \in \mathtt{closure}(g_s)$. This problem is extremely important for both queries and updates.

In this section, we will present an algorithm that solves this problem and discuss its complexity. For this analysis, we consider D to be an extended RDF/S graph. We denote by M_t the set of triples of D: $M_t = \{(s, p, o) \mid \exists i \ \in \mathcal{I}, (s, p, o, i)\ a\ quadruple\ in\ D\}$ and with M_g the set of named graphs in D: $M_g = \{n \mid \exists i \in \mathcal{I}, n \in \mathbf{G}, g_s \in \mathcal{GS}, \mathtt{gid}(g_s) = i, n \in \mathtt{ngnms}(g_s)\}$.

Per the definition of closure for graphsets (see Definition 8, Section 4.2), it follows that in order to determine whether $t \in \mathtt{closure}(g_s)$, we must compute $\mathtt{closure}(g_s)$: this means that we must consider all possible combinations of triples in M_t that involve at least one triple from each of the named graphs in $\mathtt{ngs}(g_s)$. For instance, consider the class subsumption hierarchy shown in Fig. 4.3. To compute the closure of the graphset that is comprised of named graphs g_2, g_9 and g_3, we need to consider the *combination of all triples from all three named graphs.*

The number of triples in one named graph is bounded only by M_t, whereas the number of constituent named graphs of some graphset is bounded by M_g; thus, in the worst case, we should consider $(2^{M_t})^{M_g}$ different triple combinations.

However, there is a better way to do this calculation which avoids the aforementioned exponential complexity. The general idea is that, instead of directly calculating the contents of $\mathtt{closure}(g_s)$ (as the above approach implies), we will compute *the set of triples*

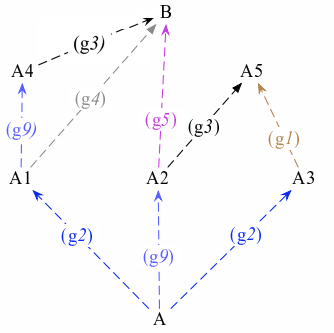


Figure 4.3: sc Hierarchy

from which t can be implied, which is *equivalent* to calculating the graphsets that t belongs to; Following this approach, we can determine whether a triple t belongs in a graphset g_s in $O(M_t \log(M_t))$ time.

It is straightforward that this is trivial for triples that cannot be implied by others: these are all the triples that *do not involve* the RDFS type, sc and sp relationships: recall that the inference rules in Tables 3.1 and 4.1 consider only those. This is achieved in $O(\log(M_t))$.

For the triples that involve the aforementioned built-in RDFS properties, a more sophisticated approach is required. In our approach, we view the sc and sp hierarchies as *directed acyclic graphs* (according to rules CFV_2 and CFV_3 of Table 3.5). Nodes in the graph are classes (properties) resp., and there exist an edge between nodes x and y iff x is a subclass/subproperty of y resp.. The problem of obtaining the sequence of triples from which triple (A, sc, B) can be implied is equivalent to discovering the path(s) (i.e., sequences of edges) in the DAG from node A to node B. For each of these sequences of triples, we are going to keep the named graphs to which each triple belongs to. The algorithm uses a depth-first search and terminates when B is reached. At each step of the algorithm, we *(i)* find the superclasses/superproperties of the context node (i.e., the node that we are currently looking at) and *(ii)* store the set of named graphs to which the triples (i.e., edges) that we have already examined, belong to.

In the above algorithm for step *(i)*, we use the labeling scheme introduced in [11] that captures the subsumption relationships and allows us to determine whether a class (or property) is a subclass (or subproperty) of another in *constant time*. This is achieved by simply comparing the labels of the classes/properties. The labeling scheme is *solely used to prune irrelevant subclass and subproperty paths*, thereby limiting our search space significantly, without taking into consideration named graph and graphset information.

For instance, consider the subclass hierarchy shown in Fig. 4.3. Suppose that we need

to discover the graphsets to which triple (A, sc, B) belongs to. We start with class A and in the first step we will keep classes A_1,A_2 and A_3 and sets of named graphs $\{g_2\}$, $\{g_9\}$ and $\{g_2\}$, since the triples that determine that A is a subclass of A_1, A_2 and A_3 are determined by triples that belong to named graphs $\{g_2\}$, $\{g_9\}$ and $\{g_2\}$ respectively. Then, for each of the obtained classes, we perform exactly the same process and we will be extending the sets of graphsets that we have obtained. The result of this process will be sets of named graphs $\{g_2, g_9, g_3\}$, $\{g_9, g_5\}$ and $\{g_2, g_4\}$. Algorithm 1 discusses more formally the above process.

Given the fact that we prune paths that are not used to imply the triple in question using the labeling scheme, and that no cycles are allowed, the described process will, in the worst case, consider each triple in the extended RDF/S graph D once. Given that each triple access requires a search in the extended RDF/S graph, the cost of each access is $O(\log(M_t))$ (using an adequate indexing system). Therefore, the total complexity is $O(M_t \log(M_t))$.

Input: Classes/Properties *source* and *target*, extended RDF/S graph D, Set of named graph names S;
Ensure: Set of named graph names S;
1: $i = 0$;
2: **for all** classes c, s.t. $\exists$ quadruple $(source, \mathsf{sc}, c, j)$ **do**
3: **if** c is a subclass/subproperty of *target* **then**
4: Set $i := i + 1$
5: Create a set of named graph names S_i
6: $S_i = S \cup N$, where $N = \mathtt{ngnms}(g_s), \mathtt{sid}(g_s) = i$
7: CALL SUBSUMPTION_IMPLICATION(c,*target*,D,S_i)
8: **else if** *source*=*target* **then**
9: return S;
10: **end if**
11: **end for**

Algorithm 1: SUBSUMPTION_IMPLICATION

For the other triples that may appear as implicit ones, the algorithm is similar. In particular, if t is of the form (P, sp, Q), then the algorithm is identical to the above, except that properties and subproperty relationships are considered. If t is of the form (A, type, B) then the algorithm is almost identical, except that, in the first step of the recursion, we search for all classes whose explicit instance is A and are also (implicit or explicit) subclasses of B; the rest is the same. Finally, if t is of the form (x, P, y), then, again, the algorithm is identical, except that, in the first step of the recursion, we search for all explicit triples of the form (x, Q, y) such that Q is an explicit or implicit subproperty of P. The rest of the recursion steps are as in the above cases.

Chapter 5

Managing Extended RDF/S graphs

In this chapter we investigate how to query and update extended RDF/S graphs. Note that extended RDF/S graphs feature a straightforward first-order representation: a relation of quadruples with the attributes *subject*, *property*, *object* and *graphset*.

5.1 Querying Extended RDF/S Graphs

In this work we extend the framework proposed in [14] to query extended RDF/S graphs.
Syntax: Let $\mathcal{V}$ be a set of variables for resources and $\mathcal{GV}$ a set of variables for graphsets. $\mathcal{V}$, $\mathcal{GV}$, $\mathbf{U}$ and $\mathbf{L}$ are mutually disjoint sets. Recall that $\mathbf{G}$ and $\mathcal{I}$ are the sets of named graph and graphset identifiers respectively.
As in [14] we rely on *tableau queries* to formalize the semantics of our query language. A tableau query is a pair (H, B) where H and B are called the *head* and *body* of the query respectively. It is often the case that a tableau query is written in the form $H \leftarrow B$ to indicate the similarity with Datalog. In our setting, H and B are quadruples featuring many-sorted variables. All variables in H occur in B. Variables are denoted with $?x$, $?y$, ... for resources and $?g_1$, $?g_2$, ... for named graphs and graphsets. Formally:

Definition 12 *A query is a tuple of the form (H, B, C) where H and B are quadruples (s, p, o, g) in $(\mathbf{U} \cup \mathcal{V}) \times (\mathbf{U} \cup \mathcal{V}) \times (\mathbf{U} \cup \mathcal{V}) \times (\mathcal{I} \cup \mathcal{GV})$ and C is a set of constraints of the form :*

1. *v op c for $v \in \mathcal{V}$, op is one of $\{=, <, >, <=, >=\}$ and $c \in \mathbf{L} \cup \mathbf{U} \cup \mathcal{V}$*
2. *v op$'$ v' for $v, v' \in \mathcal{GV}$, op$' \in \{\doteq, \prec\}$*
3. *$g = \mathtt{gs}\{n_1, n_2, \ldots, n_k\}$ where $g \in \mathcal{GV}$ and $n_i \in \mathbf{G}$*

According to the above definition, one can express constraints on resources *(1)*, on graphsets *(2)*, and finally one can specify that a graphset considered in the query is composed of a given set of named graphs *(3)*.

Example 5. *For instance, a query that asks for all instances of class Disease that are related with property associatedwith to an instance of class Cell Receptor to which dopamine bindsTo, as well as the graphset they belong to, is the following:*

$$\begin{aligned} Q_1\text{:}\ (?x, \mathsf{type}, \mathit{Disease}, ?g_{ans}) \leftarrow \\ & (?y, \mathit{associatedWith}, ?x, ?g_1), \\ & (\mathit{dopamine}, \mathit{bindsToNeurotrans}, ?y, ?g_2), \\ & (?y, \mathsf{type}, \mathit{Cell\ Receptor}, ?g_3) \end{aligned}$$

Suppose that now we want to specify the information that the instance of Cell Receptor should be defined in graphset that comprises named graph S_1 and the fact that dopamine bindsToNeurotrans is in named graph S_3. In this case, we will write the set of constraints C: $C = \{g_n^{(1)} \lessdot g_3, g_n^{(1)} = S_1, g_2 = S_3\}$

Semantics: We will now formulate the notion of answers to a query $q = (H, B, C)$ over an extended RDF/S graph D. As in [14], we rely on a *valuation function* ν assigning values to variables, i.e. $\nu : \mathcal{V} \rightarrow \mathbf{U} \cup \mathbf{L}$. A query can be thought of as a tool for obtaining new facts. If one can find the values for the variables in B (the body of the query), which satisfy the constraints in C, then one obtains H (the head of the query). We define $\nu(B)$ to be the quadruples obtained after replacing every occurrence of a variable var in B with $\nu(var)$. A valuation actually matches the variables in B with the resources in the closure of D (i.e., $Cn(D)$) while satisfying the constraints in C.

As in [14], we call a valuation ν that satisfies the constraints in C a *matching*. For each matching ν, $\nu(H)$ is obtained by replacing each variable x in H with $\nu(x)$. There can be many such matchings, so in this case, an answer to a query q is the set formed from union of $\nu(H)$, over all matchings ν.

Definition 13 *Let $Q = (H, B, C)$ be a query and D an extended RDF/S graph. A quadruple $t \in Cn(D)$ is an answer to Q if there exist a matching ν for Q, and $t \in \nu(H)$.*

Note that we do not materialize $Cn(D)$: we use Algorithm SUBSUMPTION_IMPLICATION discussed in Section 4.3.4 that decides *on the fly* whether a triple belongs to $Cn(D)$ (that is, to some graphset g_s) in a similar manner as in [27]. On the other hand, authors in [14] materialize the set of implicit triples that should be

considered during query evaluation.

Example 5 (continued). For instance, a matching ν for query (Q_1, C) is

$$\begin{aligned} A &\rightarrow \mathit{schizophrenia}, \\ g_1 &\rightarrow S_2,\ g_2 \rightarrow S_3,\ g_3 \rightarrow S_{1,2},\quad g_{ans} \rightarrow S_2 \end{aligned}$$

Given the above valuation, an answer to (Q_1, C) is quadruple $(\&\mathit{schizophrenia}, \mathsf{type}, \mathit{Disease}, S_2)$. In the example above, $S_{1,2}$ is the identifier for graphset that is comprised of named graphs S_1 and S_2.

5.2 Updating Extended RDF/S Graphs

In this section we focus on *primitive operations* of *insert* and *delete* over quadruples that take into account the RDFS semantics as defined in [23]. More precisely, we are interested in data updates of both explicit and implicit triples (for schema updates and ontology evolution readers are referred to [22]).

The semantics of each update is specified by the corresponding *effects* and *side-effects*. The effect of an insert or delete is defined over the graphset that is specified in the operation. The side-effects ensure that the resulting extended RDF/S graph continues to be *i)* valid (according to Definition 10) and *ii)* non-redundant (per Definition 11). In particular, update semantics adhere to the principle of *minimal change* [13] and thus only a minimal number of insertions and deletions need to be performed in order to restore a valid and non-redundant state of an extended RDF/S graph. The effects and side-effects of insertions and deletions are determined by the kind of RDFS predicates involved. We distinguish between *i)* (r, type, c, g) where $r \in \mathbf{U}$, $c \in \mathbf{C}$ that asserts a classification relationship between a resource r and a class c, and *ii)* (s, p, o, g) where $s \in \mathbf{U}$, $o \in \mathbf{U} \cup \mathbf{L}$ and $p \in \mathbf{P}$ that asserts property p between resources s and o. Finally, g is a graphset identifier ($g \in \mathcal{I}$). For ease of readability we are going to use the RDF/S graphset and its identifier interchangeably. In the following, in order to decide whether a triple belongs to the closure of the extended RDF/S graph D (or in other words, whether it belongs to the closure of some graphset g_s), we employ Algorithm SUBSUMPTION_IMPLICATION discussed in Section 4.3.4.

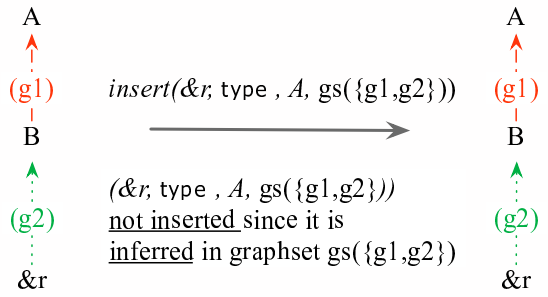
5.2.1 INSERT Operation

A primitive insert operation is of the form: $\mathsf{insert}(s, p, o, g)$ where $s, p \in \mathbf{U}$, $o \in \mathbf{U} \cup \mathbf{L}$ and $g \in \mathcal{I}$.

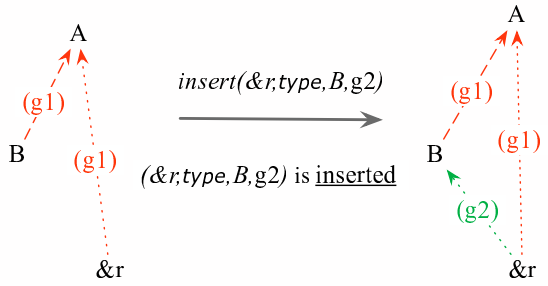
Definition 14 *We will say that an extended RDF/S graph D* satisfies *an update operation $u = \mathsf{insert}(s, p, o, g)$ iff the quadruple $(s, p, o, g) \in Cn(D)$.*

The precondition for the execution of $\mathsf{insert}(r, \mathsf{type}, c, g)$ is that c is a class in $Cn(D)$. In the positive, if (r, type, c, g) already belongs to $\mathtt{closure}(g)$, then the quadruple is not added due to the redundancy elimination rules of Table 4.4 (see Fig. 5.1(a)). Otherwise it is inserted (Fig. 5.1(b)).

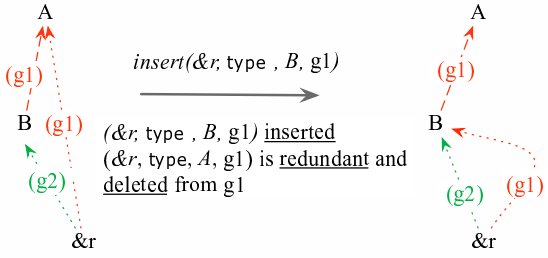
It holds as a sideffect to the above insert operation that for every quadruple $(r, \mathsf{type}, c', g')$ in D where there exists a quadruple $(c, \mathsf{sc}, c', g'')$ in $Cn(D)$ such that $ngnms(g')$=$ngnms(g) \cup ngnms(g'')$, the quadruple $(r, \mathsf{type}, c', g')$ is deleted since it can be derived from the newly added triple (according to the redundancy elimination rules of Table 4.4 (see Fig. 5.1(c)).



(a) Insertion 1



(b) Insertion 2



(c) Insertion 3

Figure 5.1: Class Instance Insertion

The preconditions for the execution of the primitive update operation $\mathsf{insert}(s, p, o, g)$ where $p \neq \mathsf{type}$ is that $s \in \mathbf{U}$, $p \in \mathbf{P}$, $o \in \mathbf{U} \cup \mathbf{L}$ and $g \in \mathcal{I}$. The semantics of the specific operation is defined in a similar manner as in the previous case. Moreover, the validity constraints CV_2 and CV_3 in Table 3.6 should be satisfied: if $(p, \mathsf{domain}, c_1, g_3)$ and $(p, \mathsf{range}, c_2, g_4)$, then $(s, \mathsf{type}, c_1, g_1)$ and $(o, \mathsf{type}, c_2, g_2)$ should hold otherwise, the operation fails.

Example 6. *For instance, suppose that we would like to link &cardiac_fibrosis (which is an instance of class Disease) through property associatedWith with &schizophrenia (i.e., insert triple (&cardiac_fibrosis,associatedWith,&schizophrenia)). This operation will fail, since &cardiac_fibrosis is not an instance of class Receptor, that is constraint CV_2 in Table 3.6 is violated.*

A formal description of the insertion of a triple to a graphset along with its sideffects can be found in Algorithm 2. Consider inserting the quadruple (x, p, y, g). At line 1 we check to see if the quadruple already exists in the closure of the extended RDF/S graph. If it does, there is no need to insert it. Otherwise, we check to see if the triple being added is a class instantiation or a property instantiation (line 2). In the first case, we ensure that the target class type exists (lines 3-4) and then we remove all existing class instances from the extended RDF/S graph that can be inferred using the new quadruple and existing class subsumption relationships (lines 5-7). Finally, we add the new quadruple (line 8). If the quadruple being inserted is a property instantiation, first we ensure that resource x is of a type compatible to the domain of p as well as y to the range of p respectively (lines 11-13). Subsequently, we remove all quadruples that can be inferred once we have added the new quadruple (lines 14-16). Finally, we add the new quadruple to the extended RDF/S graph.

5.2.2 DELETE Operation

A primitive delete operation is of the form: $\mathsf{delete}(s, p, o, g)$ where $s, p \in \mathbf{U}$, $o \in \mathbf{U} \cup \mathbf{L}$ and $g \in \mathcal{I}$.

Definition 15 *We will say that an extended RDF/S graph D* satisfies *an update operation $u = \mathsf{delete}(s, p, o, g)$ iff the quadruple $(s, p, o, g) \notin Cn(D)$.*

As with the insert operation, the precondition for the execution of $\mathsf{delete}(r, \mathsf{type}, c, g)$ is that c is a class in $Cn(D)$. In the positive, if (r, type, c, g) does belong in $Cn(g)$, then the quadruple is deleted. If there exist quadruples $(r, \mathsf{type}, c', g')$ and $(c', \mathsf{sc}, c, g'')$ with $g = \mathsf{gs}(\{g', g''\})$, then quadruple $(r, \mathsf{type}, c', g')$ is deleted since otherwise (r, type, c, g)

Data: insert(x, p, y, g), Extended RDF/S graph D
Result: Updated Extended RDF/S graph D

if $\exists\ (x, p, y, g) \in Cn(D)$ **then return** D;
if $p = \mathsf{type}$ **then**
  if $y \notin \mathbf{C}$ **then**
    return D;
  forall $(x, \mathsf{type}, z, g_2) \in D$ *s.t.* $\exists (y, \mathsf{sc}, z, g_3) \in Cn(D)$ *and* $g_2 = \mathsf{gs}(\{g, g_3\})$ **do**
    $D = D \setminus \{(x, \mathsf{type}, z, g_2)\}$;
  end
  $D = D \cup \{(x, p, y, g)\}$;
  return D;

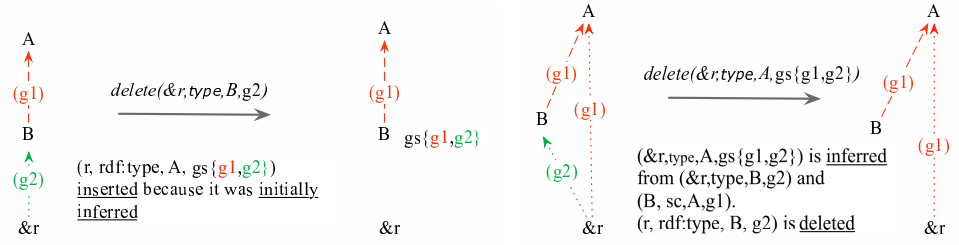
else if $\nexists\ (p, \mathsf{domain}, X, g_i) \in D$, $(x, \mathsf{type}, X, g_j) \in Cn(D)$ *or* $\nexists\ (p, \mathsf{range}, Y, g_k) \in D$, $(y, \mathsf{type}, Y, g_l) \in Cn(D)$ **then**
  return D;

forall $(x, p_2, y, g_2) \in D$ *s.t.* $\exists (p, \mathsf{sp}, p_2, g_3) \in D$ *and* $g_3 = \mathsf{gs}(\{g, g_2\})$ **do**
  $D = D \setminus \{(x, p_2, y, g_2)\}$;
end
$D = D \cup \{(x, p, y, g)\}$;
return D;

Algorithm 2: Class and Property Instance Insertion

would still be implied from the existing quadruples through the rules of Table 4.1 (see Fig. 5.2(b)).

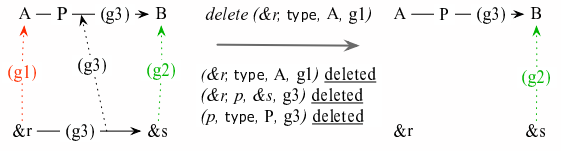
One of the sideffects of a delete operation aims to minimize the changes made to those described by the update. Therefore, we must retain the quadruples that were implied by at least one of the rules of Table 4.1: considering rule RR_4, We insert all quadruples $(r, \mathsf{type}, c', \mathsf{gs}(\{g, g'\}))$ to D where there exists a quadruple (c, sc, c', g') in D, (see Fig. 5.2(a)). By inserting only those quadruples which were associated to the deleted quadruple through a direct sc link we avoid creating redundancies which would have to be resolved as well.



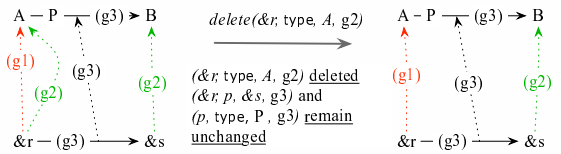
(a) Deletion 1 (b) Deletion 2

Figure 5.2: Class Instance Deletion

In addition to the side effects related to the classification links, we must take into account side-effects related to properties. More specifically, we need to ensure that the graph obtained still satisfies validity constraints CV_2 and CV_3 depicted in Table 3.6. If for instance, after the deletion, there exist a quadruple (s, p, o, g) and there does not exist quadruple $(s, \mathsf{type}, c, g_1)$ where c is the domain of p. Therefore, we have to delete quadruple (s, p, o, g_3) to ensure constraint CV_2 (see Fig. 5.3(a)). The same should happen if there does not exist quadruple $(o, \mathsf{type}, c', g_4)$ where c' is the range of p to ensure CV_3. Otherwise no more side effects take place (see Fig. 5.3(b)).



(a) Deletion 1



(b) Deletion 2

Figure 5.3: Class Instance Deletion Effects and Side-effects

The semantics of the operation $\mathsf{delete}(s, p, o, g)$ where $p \neq \mathsf{type}$ are defined as follows: essentially quadruple (s, p, o, g) denotes that there exist an edge between resources s and o that is an instance of property type p. Deleting (s, p, o, g) means that the instantiation link between edge and property type p is deleted.

If there is no quadruple (s, p, o, g) in $Cn(D)$, then the operation is void. Otherwise, we delete all quadruples (s, p', o, g') in the extended RDF graph D such that $(p', \mathsf{sc}, p, g'')$ in $Cn(D)$.

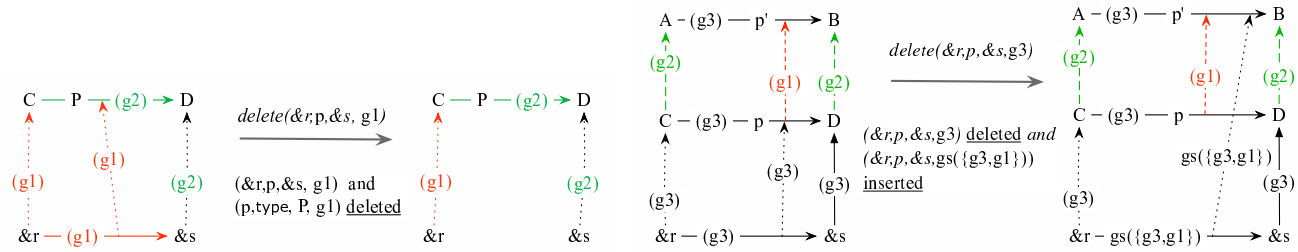
As with class instances we aim to preserve as much information as possible. Therefore, as a sideffect,we reinsert all quadruples $(s, p'', o, gs(\{g, g''\}))$ to D where there exists a quadruple $(p, \mathsf{sp}, p'', g'')$ in D. The reinserted quadruple, along with all the hierarchy tree above it are maintained because they are implied from rule $I_g^{(6)}$ in Table 4.1 (see Fig. 5.4(b)).

Now we will make a detailed description of a deletion of a triple from a graphset along with its sideffects in accordance to Algorithm 3. Consider deleting a triple $(x,\ p,\ y)$ from a graphset g, and p is the property 'type'. First we find all inferred quadruples that will not continue to be inferred once the necessary, for the deletion, quadruples are removed and

Data: $\mathsf{delete}(x, p, y, g)$, Extended RDF/S graph D
Result: Updated Extended RDF/S graph D

if $p = \mathsf{type}$ **then**
  forall $(x, \mathsf{type}, y', g_1) \in Cn(D)$,$(y', \mathsf{sc}, y, g_2) \in Cn(D)$ *s.t.* $g = \mathsf{gs}(\{g_1, g_2\})$ **do**
    forall $(y', \mathsf{sc}, z, g_3) \in D$ *s.t.* $y' != z$ **do** **if** $\nexists(z, \mathsf{sc}, y, h) \in Cn(D)$ *s.t.* $g2 = \mathsf{gs}(\{h, g3\})$ **then**
      $D = D \cup \{(x, \mathsf{type}, z, \mathsf{gs}(\{g_1, g_3\}))\}$;
  end
  forall $(x, \mathsf{type}, y', g_1) \in D$,$(y', \mathsf{sc}, y, g_2) \in Cn(D)$ *s.t.* $g = \mathsf{gs}(\{g_1, g_2\})$ **do**
    $D = D \setminus \{(x, \mathsf{type}, y', g_1)\}$;
  end
  if $\nexists\,(x, \mathsf{type}, y, g')$, *for some* g' **then**
    forall (x, p_i, o_i, g_i), $(p_i, \mathsf{domain}, d_i, g_i') \in D$ **do**
      if $\nexists(x, \mathsf{type}, z, g_j) \in Cn(D)$ *s.t.* $z = d_i$ **then**
        if $\nexists(x, q_i, o_i, h_i)$, $(p_i, \mathsf{domain}, d_i', h_i') \in D$ *s.t.* $\nexists(x, \mathsf{type}, z', h_j') \in Cn(D)$ *s.t.* $z' = d_i$ *and* $\exists(p_i, \mathsf{sp}, q_i, h_j') \in Cn(D)$ **then**
          $\mathsf{delete}(x, p_i, o_i, g_i)$;
    end
    forall (s_i, p_i, x, g_i), $(p_i, \mathsf{range}, r_i, g_i') \in D$ **do**
      if $\nexists(x, \mathsf{type}, z, g_j) \in Cn(D)$ *s.t.* $z = r_i$ **then**
        if $\nexists(s_i', q_i, x, h_i)$, $(p_i, \mathsf{range}, r_i', h_i') \in D$ *s.t.* $\nexists(x, \mathsf{type}, z', h_j') \in Cn(D)$ *s.t.* $z' = r_i$ *and* $\exists(p_i, \mathsf{sp}, q_i, h_j') \in Cn(D)$ **then**
          $\mathsf{delete}(s_i, p_i, x, g_i)$;
    end
  return D;
else
  forall $(x, p', y, g_1) \in Cn(D)$,$(p', \mathsf{sp}, p, g_2) \in Cn(D)$ *s.t.* $g = \mathsf{gs}(\{g_1, g_2\})$ **do**
    forall $(p', \mathsf{sp}, q, g_3) \in D$ *s.t.* $p' != q$ **do** **if** $\nexists(q, \mathsf{sp}, p, h) \in Cn(D)$ *s.t.* $g2 = \mathsf{gs}(\{h, g3\})$ **then**
      $D = D \cup \{(x, q, y, \mathsf{gs}(\{g_1, g_3\}))\}$;
  end
  forall $(x, p', y, g_1) \in D$,$(y', \mathsf{sp}, y, g_2) \in Cn(D)$ *s.t.* $g = \mathsf{gs}(\{g_1, g_2\})$ **do**
    $D = D \setminus \{(x, p', y, g_1)\}$;
  end
  return D;

Algorithm 3: Class and Property Instance Deletion



(a) Deletion 1 (b) Deletion 2

Figure 5.4: Property Instance Deletion Effects and Side-effects

add them if possible. More specifically, we find all instances of the deleted resource that are of a subclass type to that being deleted(line 2). For each of these we find their direct parents that are not descendants of the class from which we are deleting and explicitly classify the resource x under them (lines 3-5). Next we find the instantiation quadruples responsible for the existence of the quadruple being deleted and we remove them (lines 7-9). Next we need to check if the resource is still of the class we are removing it from in some other graphset (line 10). If not, then we must also remove all property instances that involved x as subject or object (lines 11-19). Therefore, we find all properties, whose domain is of a subclass type to y, and x is the subject (line 11) and delete them (line 13) using a call to *ruldelete*, however, this time referring to a property. This is useful because *ruldelete* will tend to any minimal change sideffects that might be necessary. The recursion is bound to two steps as a property instance deletion cannot cause any other deletions. However, a problem that occurs is that if we delete all property instances regardless of whether they are related through a hierarchy, depending on the order of the deletions we may well reinsert quadruples we wanted removed. Therefore, we filter the deleted property instances and keep only those that are highest in the hierarchy. That is, those that do not have any ancestors that will be deleted (line 12). This is safe since by removing an instance from higher in a subclass hierarchy we also remove it from all respective descendants. The whole process is repeated for the case in which x is the object of some property instance (lines 16-20). If p is a property then a similar procedure is followed as with class instances. First we explicitly add all quadruples that can be maintained (lines 23-26) and then removed the desired quadruple (lines 28-30).

Chapter 6

RUQL with RDF/S Named Graphs

In this chapter we will present in detail how the syntax and the semantics of the declarative languages RQL and RUL as they have been extended to support the notion of named graphs according to the querying and update semantics described in Chapter 5.

6.1 RQL with Graphsets

RQL (RDF Query Language) [19] is a typed query language following a functional approach (*à* la OQL [10]). It is defined by a set of basic queries and iterators which can be used to build new ones through functional composition. RQL supports sophisticated patten patching facilities, featuring variables on labels for both nodes (i.e., classes) and edges (i.e., properties).

The RQL language has three basic clauses. The `SELECT` clause is used to declare what will appear in the result. This can be a constant, a variable or even an entire RQL subquery. The `FROM` clause is used to declare path expressions. Path expressions are expressions used for pattern matching and RDF/S path traversal. The `WHERE` clause allows the declaration of filtering expressions on variables declared in the `FROM` clause.

To help describe our language we first present a first order logic view of our extended RDF/S graph. We will need a single predicate, named *quad*, which has arity four (i.e., it is of the form $quad(s, p, o, g)$), which can capture all triples used in an RDF graph, as well as the graphset they belong to. More specifically, this predicate essentially relates each triple with the graphset that it belongs to. The first element (s) corresponds to the subject of a triple, the second (p) to the predicate, the third (o) to the object and finally the last one (g) corresponds to the graphset to which the triple ($s,\ p,\ o$) belongs. Formally however, g

will actually correspond to the set of named graph URIs which are returned by the *name* function as defined in Section 4.3.

6.1.1 Subclass and Subproperty Transitivity and Type Inference

Since our language takes full advantage of RDFS reasoning we will add the respective RDFS inference rules to our FOL representation of an RDF graph as follows:

- $quad(x,\texttt{typeof},y,g) \wedge quad(y,\texttt{subclassof},z,h) \rightarrow quad(x,\texttt{typeof},z,g \cup h)$
- $quad(x,\texttt{subpropertyof},y,g) \wedge quad(y,\texttt{subpropertyof},z,h) \rightarrow quad(x,\texttt{subpropertyof},z,g \cup h)$
- $quad(s,p,o,g) \wedge quad(p,\texttt{subpropertyof},q,h) \rightarrow quad(s,q,o,g \cup h)$

6.1.2 Path Expressions

RQL *path expressions* are *path elements* or combinations of path elements. These path elements are presented in Table 6.1.2 along with their translation into logic. In our extention of the language every path element can be prefixed with a graphset usage operator "::" which will acquire the graphset information from the path element. A graphset variable can be constrained in the `WHERE` clause or can be replaced by a constant. However, for the use of a constant graphset, an alias for that constant must be declared in the `USING NAMEDGRAPH` clause. Variables starting with $ are class variables. Variables starting with @ are property variables. Any class or property variable can be replaced by constant class or property names (respectively). All other variables are either resources or literal values. These cannot be replaced by constants in a path expression but can be filtered in the `WHERE` clause. Class and property variables or constants can be prefixed with the "^" symbol. This is the *strict operator* and when used, the query process will disregard all inference rules, returning only results in the explicit part of the ontology. For instance, pattern P2 in Table 6.1.2 will collect all class instances X along with the class $\$C$ of which they are an instance and the graphset to which the respective *typeof* triple belongs to. Pattern P8, will collect all pairs of resources X, Y that are related through a property $@P$ as well as the graphset to which the property instance triple belongs.

Two or more path elements can be combined together using a '`.`'. This is a syntactical shortcut that allows easy path traversal of an RDF graph. We have added an extra syntactical convenience so that when a graphset variable is specified in the begining of a composite path expression resulting graphset will be the union of the graphsets of all connected path elements as seen in Table 6.1.2. If the same graphset variable is used explicitly in two or more elements of a path composition, then the graphset results from

Table 6.1: RQL Path elements

Pattern	Translation
P1: $g :: \$C$	$quad(c,\texttt{typeof},\texttt{Class},g)$
P2: $g :: \$C\{X\}$	$quad(c,\texttt{typeof},\texttt{Class},h)$, ($quad(x,\texttt{typeof},c,g)or$
P3: $g :: \$C\{X;\$D\}$	$quad(d,\texttt{subclassof},c,h)$, $quad(x,\texttt{typeof},d,g)$
P4: $g :: \$C\{\$D\}$	$quad(d,\texttt{subclassof},c,g)$
P5: $g :: @P$	$quad(p,\texttt{typeof},\texttt{Property},g)$
P6: $g :: \{\$C\}@P\{\$D\}$	$quad(p,\texttt{typeof},\texttt{Property},g)$, $quad(p,\texttt{domain},c,h)$, $quad(p,\texttt{range},d,i)$
P7: $g :: \{X;\$C\}@P\{Y;\$D\}$	$quad(x,p,y,g)$, $quad(p,\texttt{domain},c,h)$, $quad(p,\texttt{range},d,i)$
P8: $g :: \{X\}@P\{Y\}$	$quad(x,p,y,g)$
P9: $g :: @P;@Q$	$quad(p,\texttt{subpropertyof},q,g)$

Pattern	Translation
P10: $g :: \{X\}@P\{Y\}.h :: @Q\{Z\}$	$quad(x,p,y,h), quad(x,p,y,i)$
P11: $g :: \{X\}@P\{Y\}.@Q\{Z\}$	$quad(x,p,y,h), quad(x,p,y,i) where\ g = h \cup i$

Table 6.2: Composing path elements

each element will be tested for equality and only those pairs that are equal are kept as results.

6.2 Declaring and using named graphs and graphsets

6.2.1 `USING NAMEDGRAPH` clause

This clause is used to define constant graphsets which can be used instead of graphset variables in path expressions or to update operations to a specific graphset. Here is an example of its use:

```
SELECT x
FROM g::Resource{x}
USING NAMEDGRAPH g=(&http://biology#IUPHAR,&http://biology#GSK)
```

This query will return all individuals which have been declared as an instance of the class resby the named graph IUPHAR, by the graph GSK or jointly by both. To constrain results to those that are strictly shared by both named graphs in the previous example we would rewrite it using the strict operator '^' as follows:

```
SELECT x
```

```
FROM g::Resource{x}
USING NAMEDGRAPH g=^(&http://biology#IUPHAR,&http://biology#GSK)
```

Essentialy, if **N**is the set of named graphs we are querying, using '^' we will access triples in $\Gamma(\mathbf{N})$. Where as without it, it will also involve triples in $\Gamma(\subset \mathbf{N})$

6.2.2 Graphset Operations

We have added a new graphset construction function to the language named `gset()`. It similar in nature to the existing bag constructor operations. The operation takes as parameters any number of named graphs (i.e. resource URIs) and returns the corresponding graphset. For example:

```
SELECT gset(g,&http://biology#IUPHAR)
FROM Graph{g}
```

Furthermore, we have overloaded some existing operators of the language to include graphsets. Here is a list of these operators:

'=' This operator implements the equality relation defined in section 4.2.

'in' A named graph URI is constituent of a graphset if one as described in section 4.2 (i.e. $g_n \lessdot g_s$.

'<' This binary operator between graphsets returns true if every named graph in the former graphset is also a named graph of the latter. False otherwise.

'>' This binary operator between graphsets returns true if every named graph in the latter is also a named graph of the former. False otherwise.

6.2.3 The DEFAULT# Graphset

It is not unlikely that in many cases some triples inserted into a RDF database are not associated with a specific named graph. In this case, to maintain consistency with our model, we associate all such triples to a specific built-in named graph with the URI `DEFAULT#`. This also allows for backward compatibility with previous versions of the language that didn't support named graphs. Unlike user defined named graphs the default graph is not a resource and can not participate in properties as subject or object and cannot be removed.

6.3 RUL with Graphsets

RUL (RDF Update Language) is an extention language over RQL used for updating RDF graphs [23]. RUL ensures that the execution of the update primitives on nodes and arcs

neither violates the semantics of the RDF model nor the semantics of the given RDFS schema. In addition, RUL supports fine-grained updates at the class and property instance level, set-oriented updates with a deterministic semantics and takes benefit of the full expressive power of RQL for restricting the range of variables to nodes and arcs of RDF graphs. The language has been implemented in the context of RDF Suite, as an extension of RQL. The implementation relies on a database update language and generates SQL update statements for the various database representations used in RDF Suite.

We have enhanced the RDF Update Language by extending the insert and delete update operations with a graphset specifier. This specifier denotes to which named graph or set of named graphs a piece of information must be added or from which named graph or set of named graphs it must be deleted. The graphset specifier can be a constant declared in the `USING NAMEDGRAPH` clause of a variable defined in the `FROM` clause. Below, we will present the semantics of each individual operation.

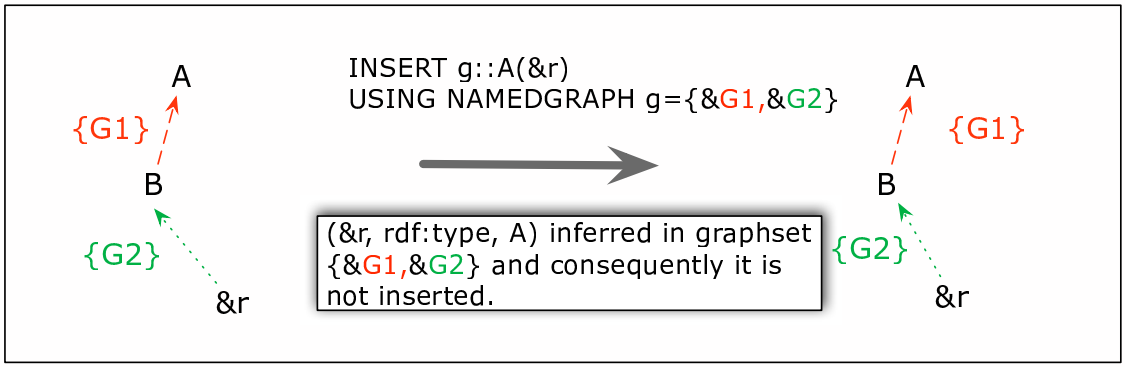
6.3.1 Class instance insertion to a graphset

The syntax of the INSERT statement for class instances in graphsets is as follows:

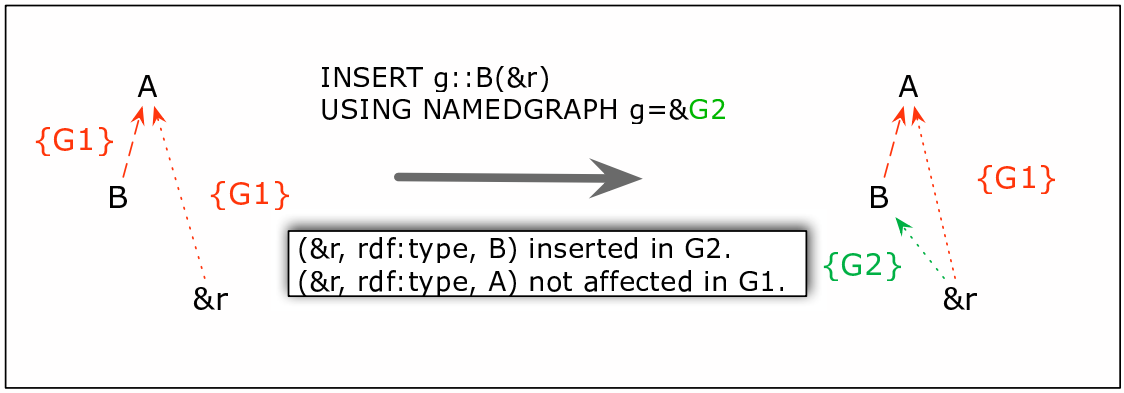
```
INSERT [GraphsetExp::]QualClassName(ResourceExp)
[FROM VariableBinding]
[WHERE Filtering]
[USING NAMESPACE NamespaceDefs]
[USING NAMEDGRAPH GraphsetDefs]
```

This INSERT operation introduces new classification links described by *QualClassName* for individuals described by *ResourceExp*. If those the resources don't already exist, then the operation also creates the respective nodes. The only difference of this operator with the original RUL insertion operation, is that the new classification triple associated with the graphset *GraphsetExp* as well, indicating the graphset in which the insertion(s) should take place. The latter can be a constant declared in the `USING NAMEDGRAPH` clause or a resource or graphset variable from the `FROM` clause. If the same triple already exists in the same graphset in an inferred or explicit manner, it will not be inserted (see Figure 6.1(a)).

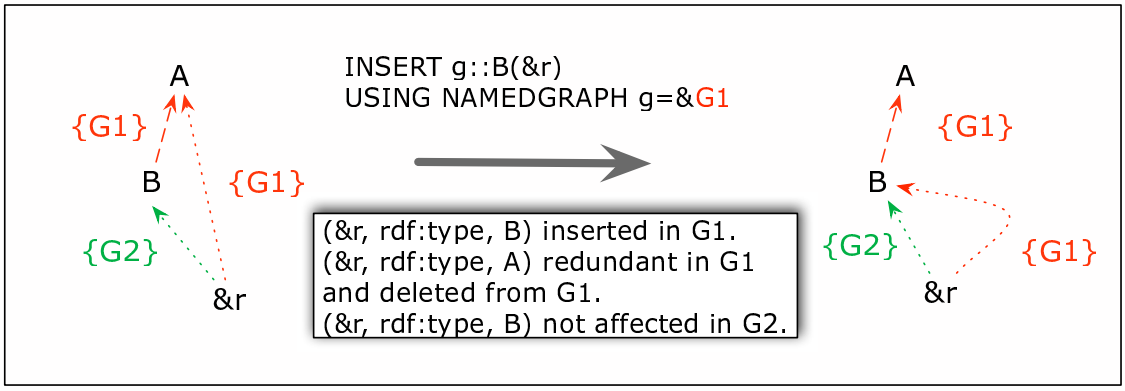
However if the same triple exists but in a different graphset, then the new classification will be inserted (Figure 6.1(b)).



(a) Insertion 1



(b) Insertion 2



(c) Insertion 3

Figure 6.1: Example class instance insertion effects and sideffects.

If a superclass instance that belonged to a graphset G can now be inferred again in G as a result of the new classification, then the former instantiation triple is removed because it is redundant (Figure 6.1(c)).

If one of the named graphs in *GraphsetExp* doesn't exist then the operation fails. The *GraphsetExp* is optional; this is done for ease of use, as well as for backward compatibility with the original RUL. More specifically, in this case we imply the use of the `DEFAULT#` graphset.

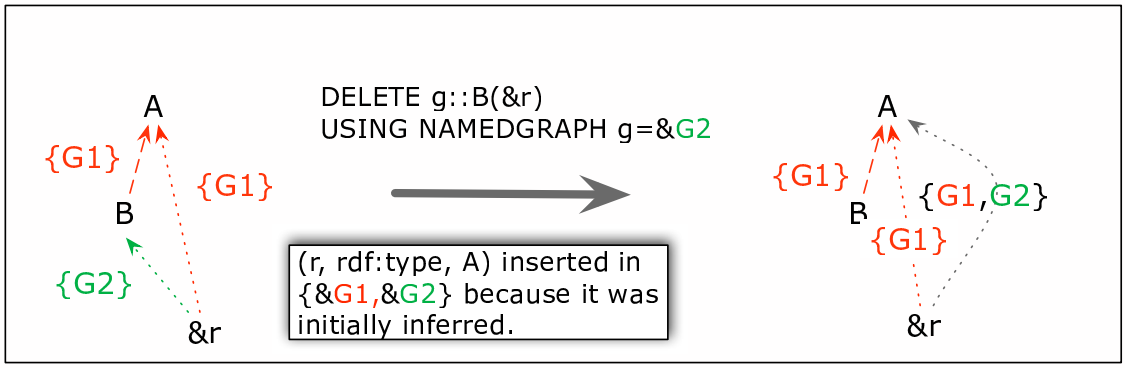
6.3.2 Class instance deletion from graphset

The syntax of the DELETE operation for class instances from graphsets is as follows:

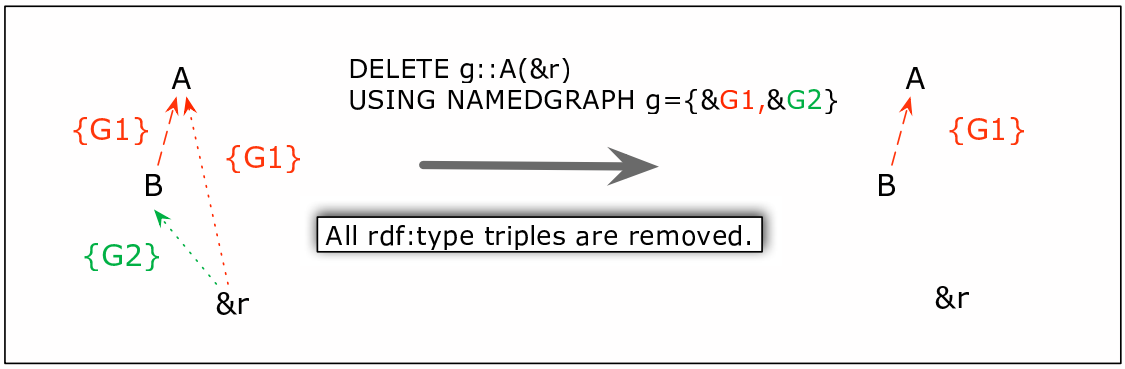
```
DELETE [GraphsetExp::]QualClassName(ResourceExp)
[FROM VariableBinding]
[WHERE Filtering]
[USING NAMESPACE NamespaceDefs]
[USING NAMEDGRAPH GraphsetDefs]
```

The DELETE operation erases classification arcs and possibly also nodes from an RDF graph. Again, the difference with the DELETE operation appearing in the original RUL is that here we can specify from which graphset the classification triples are to be removed. The expression *ResourceExp*, which denotes the node from which the classification arc to be deleted originates, can be a URI or a variable. The effect of the DELETE operation is to remove the direct or indirect classification arc of *ResourceExp* to class *QualClassName* from the graphset *GraphsetExp* and replace it by the arc of *ResourceExp* to all the direct super-classes of QualClassName, if any. The reinsertion of the classification to the super-classes essentially aims to maintain the inferred information that will be lost along with the actual deleted classification if no further action is taken. Therefore, the replacement classifications are not inserted into *GraphsetExp* but to the graphsets to which the inferred classifications belong to (i.e., to the union of *GraphsetExp* with the graphset to which the subclass triple(s) between *QualClassName* and its super-classes belong to (Figure 6.2(a)).

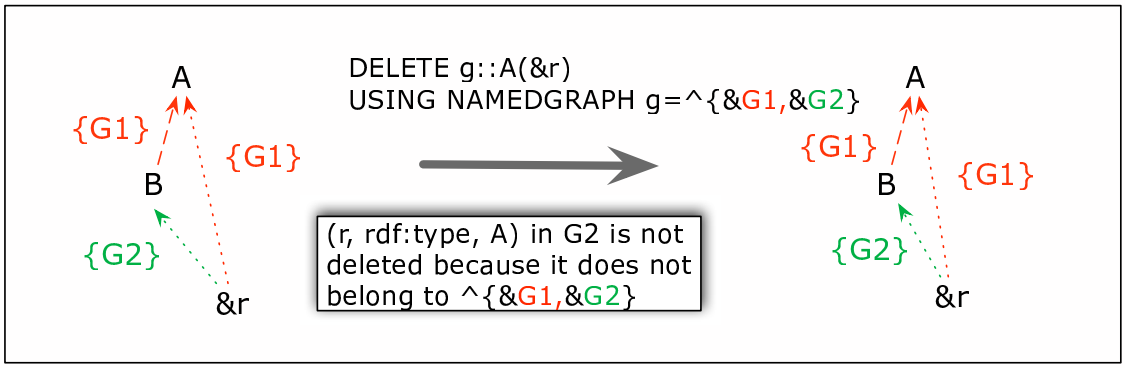
As with deletes in the original RUL, if *ResourceExp* is multiply classified, the classification arcs to classes not related to *QualClassName* remain untouched. In the original RUL, if *QualClassName* is the top of the class hierarchy (i.e., res), then the effect of the deletion is the complete removal of the resource *ResourceExp*; however, in RUL with graphsets, the effect of the operation is the deletion of the classification triple that was responsible for the



(a) Deletion 1



(b) Deletion 2



(c) Deletion 3

Figure 6.2: Example class instance deletion effects and sideffects.

inference of *ResourceExp* being a resource in the context of *GraphsetExp* only. Therefore, in the case of DAG class hierarchies, or multiple classifications of resources, where the inferred triple classifying *ResourceExp* as res may well belong to more than one graphset, we opt to minimize change to only those that are necessary to stop *ResourceExp* being an instance of res in the context of *GraphsetExp*. If, on the other hand, the path deleted is the only one, then the node *ResourceExp* is also removed.

If *GraphsetExp* is defined as a non-strict graphset (i.e we do not use '^' in front of the graphset declaration), then the delete operation applies to *GraphsetExp* as well as to all its subsets (see Figure 6.2(b)), as a syntactical convenience.

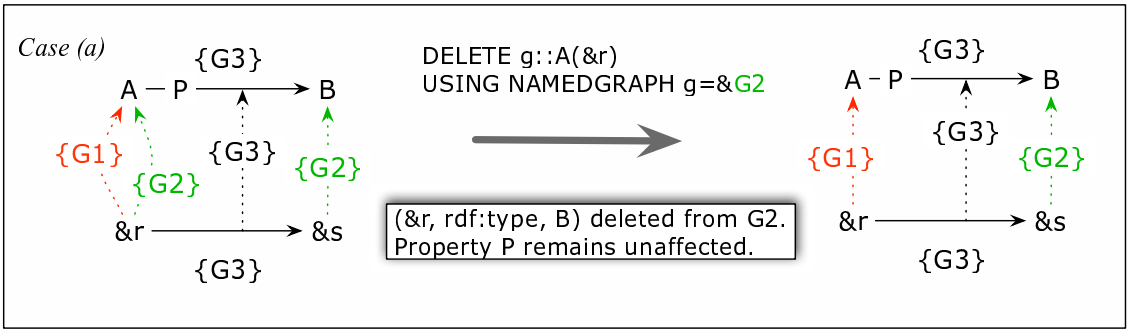
All classification arcs that are added by a DELETE operation must take the semantics of INSERT into account, so that the resulting graphsets remain without redundancies (as defined in 4.3.3). The side effects of DELETE in any of the above cases are caused by the changes in the classification of a node. To be more specific, all property arcs emanating from (leading to) the node denoted by *ResourceExp* that have as domain (range) a class, to which *ResourceExp* is no longer an instance, are also deleted by a DELETE property instance operation. These side-effects are necessary to keep the graph consistent, since *ResourceExp* does no longer belong to the declared classification (see Figures 6.3(a) and 6.3(b)).

To maintain backward compatibility, when *GraphsetExp* is not specified in a DELETE operation the graphset containing all existing named graphs is implied. This essentially encompases all possible graphsets. This way, we can delete or reclassify a resource regardless of which graphset it may belong to.

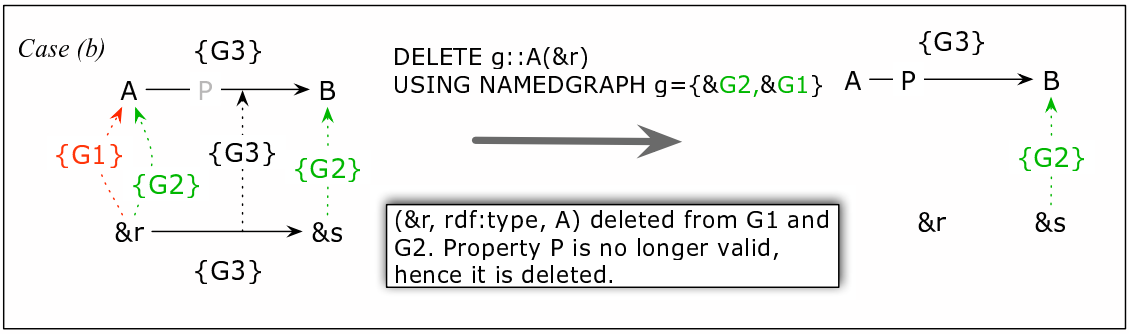
6.3.3 Property instance insertion to graphsets

The INSERT and DELETE statements can also be used to update the properties of resources i.e., arcs in an graphset. The syntax of the INSERT statement in this case is as follows:

```
INSERT [GraphsetExp::]QualPropertyName(SubjectExp, ObjectExp)
[FROM VariableBinding]
[WHERE Filtering]
[USING NAMESPACE NamespaceDefs]
[USING NAMEDGRAPH GraphsetDefs]
```



(a) Deletion 1



(b) Deletion 2

Figure 6.3: More example class instance deletion effects and sideffects.

The above INSERT operation adds to resource node *SubjectExp* a new property arc that is an instance of property *QualPropertyName* and has value *ObjectExp*. The arc (triple) that is introduced is associated with the graphset *GraphsetExp*. *SubjectExp* and *ObjectExp* can be constants or variables with bindings determined in the `FROM` clause. *GraphsetExp* can be a constant declared in the `USING NAMEDGRAPH` clause or a resource or graphset variable from the `FROM` clause. Again, RQL typing rules for triples must be respected: *SubjectExp* must evaluate to a URI, instance of the domain of property *QualPropertyName*, and *ObjectExp* must evaluate to a URI or literal value, instance of the range of property *QualPropertyName*.

If another property arc from *SubjectExp* to *ObjectExp* exists and this arc is an instance of a super-property of QualPropertyName in a graph G, then, if the new arc will allow the inference of the existing arc (in the context of G, as usual) then the respective triple (from G) will be removed. In any case, the operation's effect is the introduction of a new arc to *QualPropertyName* and its association with *GraphsetExp*. However, if *SubjectExp* and *ObjectExp* are not (direct or indirect) instances of the domain and range of *QualPropertyName* (respectively), then the INSERT operation fails. If the property arc exists as an instance of a sub-property of *QualPropertyName* in the graphset *GraphsetExp*, then the

operation has also no effect.

If *GraphsetExp* is not specified, then like in the case of class instance insertions, the *default* graphset is implied, for backward compatibility.

6.3.4 Property instance deletion

The syntax of the DELETE operation is as follows:

```
DELETE [GraphsetExp::]QualPropertyName(SubjectExp, ObjectExp)
[FROM VariableBinding]
[WHERE Filtering]
[USING NAMESPACE NamespaceDefs]
[USING NAMEDGRAPH GraphsetDefs]
```

As in the case of resources, the DELETE operation essentially removes the instantiation arc between *QualPropertyName* and the property arc from *SubjectExp* to *ObjectExp* from a graphset *GraphsetExp*. In addition, it inserts a arc from the arc to the super-properties of *QualPropertyName* under the graphsets in which they were each inferred, if any (see Figure 6.4). If there are

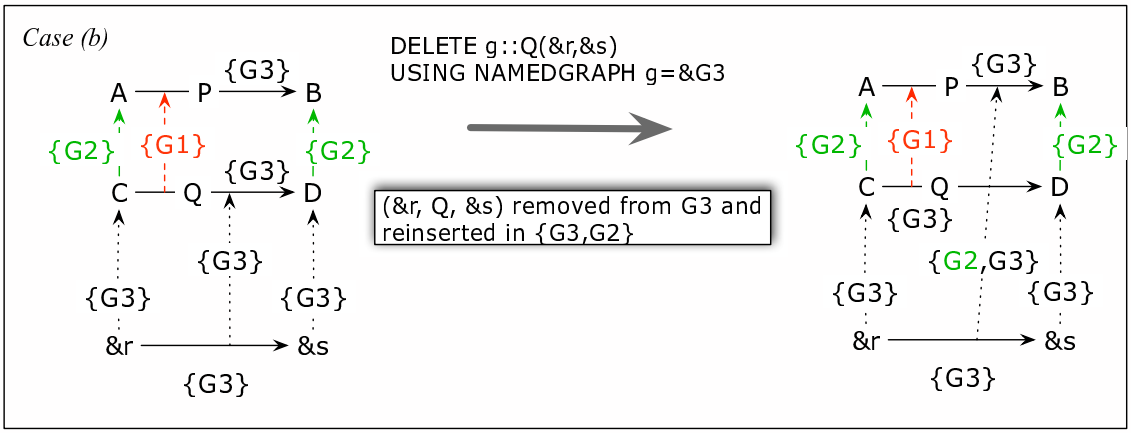


Figure 6.4: Example property deletion effects and sideffects

no super-properties of *QualPropertyName*, the arc is removed altogether. If the arc is not an instance of *QualPropertyName* in *GraphsetExp*, the operation has no effect.

If *GraphsetExp* is not specified, then the non-strict universal graphset is implied, as usual, for backward compatibility.

6.4 `REPLACE` operation

The replace operations found in the original RUL specification [23] are not considered in this thesis. It is difficult to extend the semantics of replace operations with graphsets because a replace on one graphset may affect a number of other graphsets. Moreover, in the current RUL specification, replacement of a resource URI essentially performs the replace on all instances that have that URI and all properties where it appears as subject or object. Therefore, it is challenging to describe the semantics of constraining such an operation to a single graphset.

Chapter 7

Implementation

The querying and updating of named graphs has been implemented as an expansion over the existing RQL [19] and RUL [23] languages that are a part of RDF-Suite [1]. The related persistant storage model for RDF ontologies is also extended to store information regarding named graphs.

7.1 Relational Model

In this chapter we will present the relational representation of our RDF model examined in 3.2. It is around this relational model that the query and update languages for graphsets are implemented. This is in essence a small extention over the existing representation presented in [23]

To enable the storrage of our RDF Graph with graphsets, we translate the graph representation to a triple based representation. More specifically, every directed arc in the graph representation corresponds to a triple in the triple representation where the source and target of the arc correspond to the subject and object of the triple and the label of the arc itself corresponds to the property respectively. For the introduction of graphsets and named graphs, it becomes necessary to relate information to these triples so for each relation that stores triples we add a *triple ID* collumn which associates a unique ID to each unique triple.

7.1.1 RDF Schema Representation

One type of information the relational model must store is schema information. This information includes, class and property types, IsA relationships between classes and between properties, as well as domain and range type definitions for properties. The relations used to represent schema information are presented in figure 7.1.

The relation `Classes` holds triple information for all class types. More specifically it

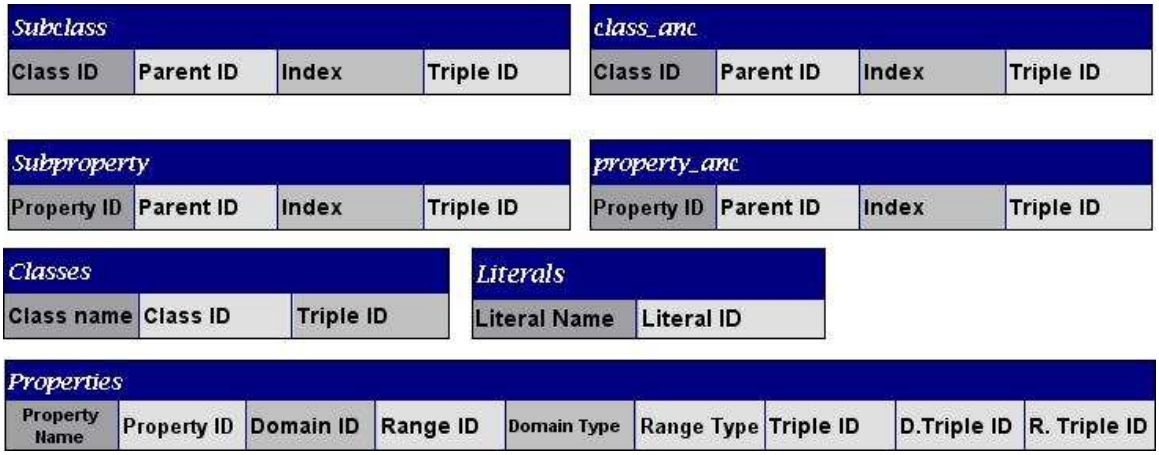


Figure 7.1: Relations used for schema representation.

stores the the class name as well as the ID of the triple instantiating the class as a type of *rdfs:class*. Relation `Properties` stores details regarding property types. These details include, the property name, the domain class name and range class name, the domain and range metaclass types as well as triple IDs for the property, domain, and range triples that are associated with this triple. The relation `Literals` contains all literal types along with their ID. [1]. To represent the subsumtion relationships between classes and between properties we use the `subclass` and `subproperty` , relations respectively. These tables contain the ID of each class or property, the id of their direct parent, the index used for fast subsumtion checking as well as the triple id for the *subclassof* or *subpropertyof* triple encompassed by each tuple. The labels held by the ID and index of classes or properties only have the ability to capture subpsumtion relations when they are in the form of a tree. However, often a schema may form a Direct Acyclic Graph (DAG). This happens when a node has more than one direct father. To capture relationships that do not belong to the spanning tree of the DAG we use the `class_anc` and `property_anc` relations.

7.1.2 RDF Data Representation

The original Relational model for RQL and RUL has three different relational representations for RDF Data (i.e. class and property instances). Each one has its advantages and disadvantages. However, in our implementation of the named graph query and update language, we only extend the Hybrid [39] representation that has better performance in most cases. The relations used to represent data information in the Hybrid representation are abstractly depicted in Figure 7.2.

[1] Class types are actually stored in the relation `1000000000` since 1000000000 is the metaclass id of the type `rdfs:class`. Respectively, property types are stored in `t2000000000`. Finaly, literal types are stored analogously in `t12`



Figure 7.2: Relations used for the representation of data.



Figure 7.3: Relations used for the representation named graphs, graphsets and their association with triples.

Table `class_instances` holds all class instances together in one relation. This relation stores the URI of the instance, the class of which the resource is an instance as well as the triple ID of the instantiation (typeof) triple. In the case of properties instances are broken in to groups of relations based on the type of their domain and their range. However for simplicity we will consider a table `property_instances` where all property instances are stored [2].

7.1.3 Graphset Representation

To introduce the association of triples with graphset into our representation we add three new relations visualised in 7.3. The relation `namedgraphs` associates all namedgraph uris to a unique ID. Note that the uris of named graphs are also found in the instance table `class_instances`. This is because named graphs are also resources and instances of the class rdfsuite:Graph. The relation `graphsets` assigns a unique ID to each graphset that is currently used explicitly is the ontology. Finally, the relation `graphtriples` associates triples IDs with graphset IDs. This is a many-to-many association. In other words a triple may belong to more than one graphset and a graphset can contain more than one triple.

7.2 Extending the RQL Interpreter

The architecture and graphset extention of the RQL interpreter can be seen in Figure 7.4. The interpreter is composed of five basic modules. Lexical and syntactical analysis is responsible for parsing queries and updates. Graph construction translates the syntax tree to an execution graph. The type inference module is responsible for the typing of values, variables and operations. Graph evaluation is responsible for executing the execution graph and finally the Database API incorporates SQL translations and acts as an interface with the underlying persistant storage.

[2]The actual implementation uses different names for the instance relations. Class instances are stored in tc20000000000 since 2000000000 is the id of the class type Resource. Property instances are stored in tables named tp`X`k`Y` where `X` the property domain kind and `Y` the property range kind

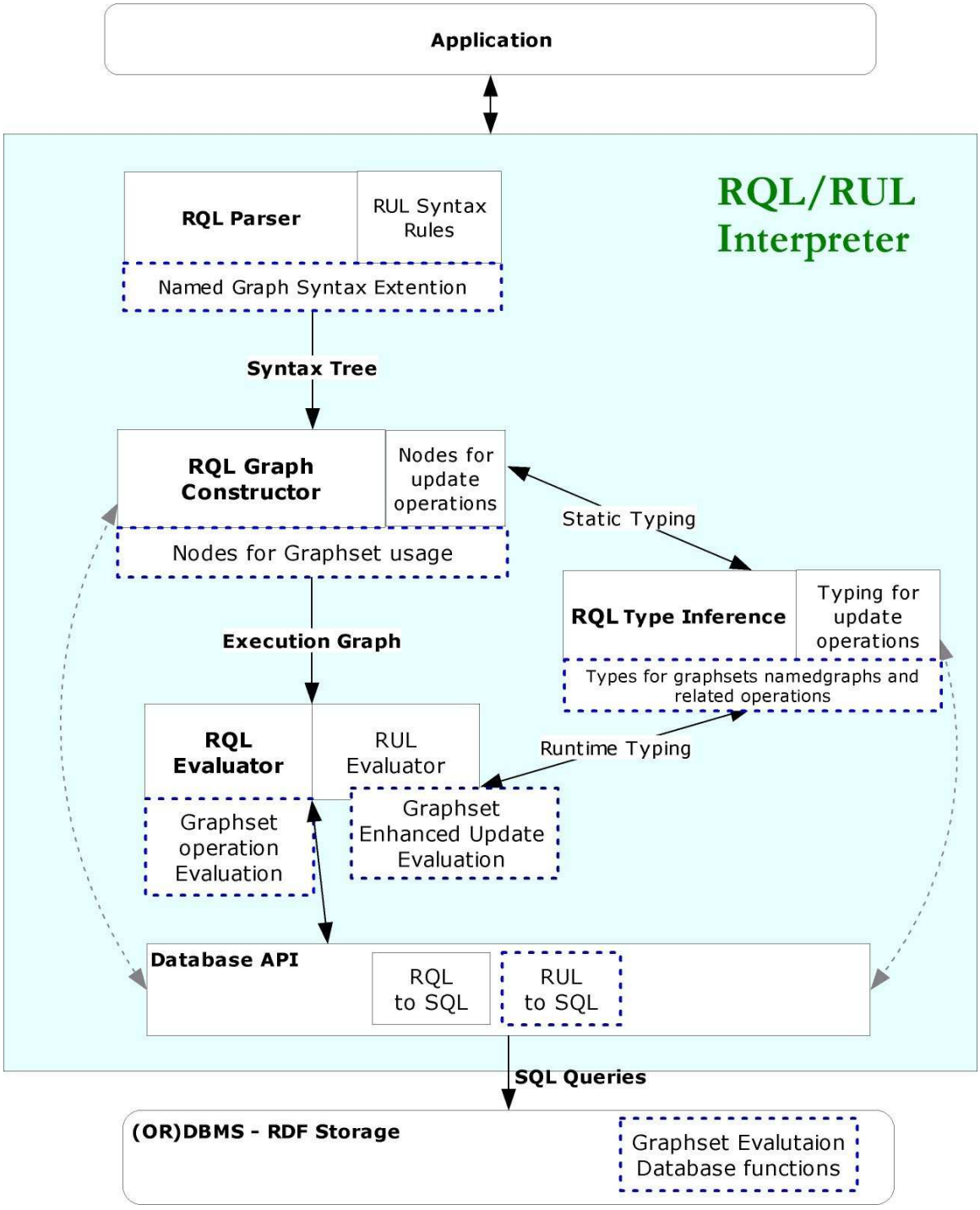


Figure 7.4: Overview of the architecture of the RQL interpreter and its extention for named graphs.

7.2.1 Syntax

Extending the syntax of the RQL Interpreter is very simple. RQL uses *Lex* for lexical analysis and *Yacc* for parsing. Support for named graphs simply required the addition of some lexical tokens and some parsing rules for the extention of update and path expressions with graphset constraints and variables. Moreover parsing rules were added for the declaration of constant graphsets through the `USING NAMEDGRAPH` clause. A full BNF specification of the grammar used by our implementation can be found in Appendix A. Through the set of declared parsing rules the syntactical analysis process builds the syntax tree used by the Graph Construction module. Path expressions and path compositions are now placed under a graphset usage node.

7.2.2 Graph Construction

Graph construction is essentially the procedure that takes as input the syntax tree from the parser and builds the execution graph. Every *select-from-where* statement in the syntax tree translates to an algebraic iterator in the execution graph. All the path expressions used in the *select-from-where* statement are represented by *scan operations* that belong to the respective algebraic iterator. A detailed analysis of the the graph construction module can be found in [18]

Graphset constraints and variable results are achieved in the following manner. When a scan operation is created, we also create an operation called `tripleids` and its parameters are the results from the scan operation. The function of this operation is to find all possible sets of triple ids that are involved in the inference of the information returned by the scan. The result of this operation is a bag of integer arrays. When the scan operation is a mixed scan (i.e. contains both data and schema variables) then the `tripleids` operation is only associated with the data triples accessed by the scan. If a path expression has a graphset usage constraint or variable attached to it, then we create a new operation named `getGraphsets`. The parameter of this operation is the `tripleids` operation assigned to the respective scan. The function of this operation is to find all the graphsets to which sets of triple ids correspond to.

In RQL, when the same variable name is used at two or more different positions in a from clause and consequently two or more different scan operations, then an equality filter between the results from each occurrence is added. This remains the same for graphset variables. There is however a syntactical shortcut in the extended language that places a slightly different filter. When a graph variable is declared at the beginning of a composite path expression and none of the internal path elements have graphset variables declared,

then the resulting graphset coresponds to the entire path. This is implemented by calculating the union of the graphsets to which each result of every path element belongs to. More specifically, we create a `joinGraphsets` operation and add as parameters to this operation, the `getGraphset` operations of each path node.

Once all operations corresponding to the `FROM` clause of the syntax tree have been created, then, wherever a graphset variable is declared, we declare a variable scan operation on the corresponding `getGraphset` or `joinGraphsets` operation. For the first scan operation of each distinct variable name we also add a name operation which will associate the variable name with the result from the variable scan.

7.2.3 Type Inference

The type inference module of RQL is responsible for validating the correctness and compatibility of parameters of operations. Moreover it associates each non-algebraic operation with its evaluation function.

To support named graphs and graphsets we introduced three new types to those handled by RQL. First of these is `QL_NAMEDGRAPH` which is essentially equivalent in nature to the type `QL_RESOURCE` as all named graphs are also resources. However, the distinction proved necessary in the implementation. Secondly we introduced two composite types `QL_GRAPHSET`, and `QL_STRICT_GRAPHSET`. Both these types are sets of `QL_NAMEDGRAPH` elements. The difference between the two graphset types is that when querying a `QL_GRAPHSET`. The result will also originate from all of its subsets. On the other hand querying a `QL_STRICT_GRAPHSET` will only return results belonging to that specific graphset. Although RQL consists a type for sets, namely `QL_SET` this was not used, as the latter type is stored in temporary tables in the database disabling flexible comparison and result handling. Although, they are resources, in our storage model named graphs have a unique numeric identifier. Therefore, when validating a named graph constant, it's numeric identifier is fetched from the database and replaces the string URI representation.

7.2.4 Basic Database Functions

In this expansion of the RQL interpreter with graphsets we made the design choice to push some basic widely used procedures to the database as stored procedures. This has the drawback of coupling the implementation of some of the procedures to the language provided by each database management system. However it also helps avoid multiple repetitive external query executions and also allows taking advantage of convenient built in types. In our algorithmic analysis we will use the following relations as an abstraction to our relational model:

1. `Classes`(*class*,*tripleid*) For every triple of the form (*class*, type, class) we assign a unique id *tripleid* to the triple and add the tuple (*class*,*tripleid*) to the relation `Classes`

2. `Properties`(*property*,*tripleid*) For every triple of the form (*class*, type, *property*) we assign a unique id *tripleid* to the triple and add the tuple (*class*,*tripleid*) to the relation `Properties`

3. `SubClass`(*child*,*parent*,*tripleid*) For every triple of the form (*child*, sc, *parent*) we assign a unique id *tripleid* to the triple and add the tuple (*child*,*parent*,*tripleid*) to the relation `SubClass`

4. `SubProperty`(*child*,*parent*,*tripleid*) For every triple of the form (*child*, sp, *parent*) we assign a unique id *tripleid* to the triple and add the tuple (*child*,*parent*,*tripleid*) to the relation `SubP roperty`

5. `ClassInstances`(*uri*,*class*,*tripleid*) For every triple of the form (*uri*, type, *class*) we assign a unique id *tripleid* to the triple and add the tuple (*uri*,*class*,*tripleid*) to the relation `ClassInstances`

6. `PropertyInstances`(*subject*,*object*,*property*,*tripleid*) For every triple of the form (*subject*, *property*, *object*) we assign a unique id *tripleid* to the triple and add the tuple (*subject*,*object*,*property*,*tripleid*) to the relation `PropertyInstances`

7. `GraphTriples`(*tripleid*,*graphset*) For every triple of the form with triple id *tripleid* that belongs to a graphset *graphset* we add the tuple (*tripleid*,*graphset*) to the relation `GraphTriples`

find class/property ancestor path involved triples This procedure shown in Algorithm 4 essentially calculates all the groups of triples that together can inform the information that one class/property type is the ancestor of another type. This is a recursive function. Its recursive nature was chosen to avoid splitting the result path whenever a DAG node is met. The procedure basically performs the transitive closure of the graph but takes advantage of the labels of each node to avoid traversing paths that will not lead to a the target class or property.

Find class/property instance involved triples This procedure shown in Algorithm 5 calculates the sets of triples that are involved in inferring a typeof relationship between a resource and a class type. To achieve this, for each instantiation triple of the target resource that instantiates a descendant class of the target class we

Input: *result*,{a bag}
anc_class, {ancestor class}
desc_class,{descendant class}
current_path {a set of triples capturing an IsA path}
Ensure: $\forall$ s $\in$ *result* : s $\vdash$ (*desc_class*,sc,*anc_class*)
1: **for all** *t* **in** *SubClass* **where** *t.class* = *desc_class* **and** *isDescendant*(*t.parent*, *anc_class*) **do**
2: *new_path* $\Leftarrow$ *current_path* $\cup$ *t.tripleid*
3: **if** *t.parent* = *anc_class* **then**
4: **add** *new_path* **to** *result*
5: **else**
6: `findAncestorPathTriples`(*bag*, *anc_class*, *t.parent*, *new_path*)
7: **end if**
8: **end for**

Algorithm 4: findAncestorPathTriples

merge that triple with the ancestor path between the target class and the actual instantiation class.

Input: *result* {a bag},
class {the class type},
uri {a resource uri}
Ensure: $\forall$ s $\in$ *result* : s $\vdash$ (*uri*,type,*class*)
1: **for all** *t* **in** *ClassInstances* **where** *t.uri* = *uri* **and** *isDescendant*(*t.class*, *class*) **do**
2: *temp_path* $\Leftarrow$ *t.tripleid*
3: **if** *t.class* = *class* **then**
4: **add** *temp_path* **to** *result*
5: **else**
6: `findAncestorPathTriples`(*result*, *class*, *t.class*, *temp_path*)
7: **end if**
8: **end for**

Algorithm 5: findInstanceInvolvedTriples

Get graphsets from triples As we have seen in previous chapters, the same triple may occur in more than one graphset. Therefore, for every path of triples calculated by the previous algorithms, there is possibly more than one sets of graphs that share the path. The role of this function (Algorithm 6) is to take a bag of triple paths and create another bag containing any graphset that to which one of the triple paths in the source bag belongs to. Again this is internally achieved with a recursive approach to avoid splitting the result when a triple belongs to more than one graphset.

Join graphsets The goal of this procedure is to take two bags containing graphsets and add to a result bag all the possible combinations of graphs from the first bag with those from the second. The combination is essentially a set theoretic union between

Input: *result* {a bag},
source_bag {a bag of sets of triples}
Ensure: $\forall$ s $\in$ *source_bag* $\exists$ g *in result* : g $\vdash$ s
1: **for all** s **in** *source_bag* **do**
2: **for all** *t* **in** s **do**
3: **for all** *g* **in** *GraphTriples* **where** *g.tripleid=t* **do**
4: **add** *g.graphset* **to** *temp_graphset*
5: **end for**
6: **end for**
7: **if** *temp_graphset not in result* **then**
8: **add** *temp_graphset* **to** *result*
9: **end if**
10: **end for**

Algorithm 6: getGraphsetFromTriples

each couple of graphsets. This operation is used when we have a path expression with many path elements that are all under the same graphset variable. The difference with performing this operation in the main memory as the scan result graphsets are returned is that we have eliminated all duplicates.

7.2.5 Execution Graph Evaluation

For each of the functions presented in the previous chapter a respective `baseopn` was added in RQL. Apart from these however, traditional, equality, less-than and other such operators were overloaded with evaluation code that allows for the comparison of graphsets. Note again that these comparison are at the level of the named graphs contained in a graphset and not the triples associated with the graphset.

7.2.6 Database API

The *Database API* part of the RQL interpreter is where the SQL code is generated and where the actual operations on the database are performed. For the query part of the graphset extention of the interpreter, few additions had to be made to the Database API. These are mostly wrapper functions for the database pprocedures defined in section 7.2.4. On the other hand for the update functions of the language, the database API code responsible for inserting and deleting class and property instance had to be rrewritten This is so because the semantics of the update operation change radically in the presence of named graphs.

Class Instance Insertion Insertion of a resource as the instance of a class under a given graphset involves two basic stages. The first step is to find and delete all class instantiations that will no longer need to be stored as they will be implied by the

new class instantiation through RDFS inference. Subsequently, we actually introduce the new instantiation link. There is no need to tend to dependant property instances as they will remain unchanged. A description of the insertion algorithm is seen in Algorithm 2 in Section 5.2.

Class Instance Deletion The deletion of a class instantiation is a much more complicated procedure. This is due to two semantical constraints. Firstly, all property instances must have as a subject and object a resource that is of equal or subclass type to that of the domain and range respectively of the property (we disregard literals for the simplicity of the explanation). This forces us to delete any depending properties that will no longer be valid after the delete. Secondly, the update process must aim to minimize changes made to the ontology to those specified by the update statement. This isn't always possible, however it is important to preserve as much irrelevant information unchanged by the delete. A detailed description of the deletion process is seen in Algorithm 3. When we delete a class instantiation from a graphset. First we find all instantiation links that are explicitly stored and that, together with the existing schema information, infer the triple to be deleted. These are the the triples that are actually going to be deleted because, as long as they exist, their inferred triples will also exist. However in the case of DAGs (i.e. classes may have more than one parent class) it is not necessary to remove the entire instantiated graph, but only the path that instantiates the resource being deleted with the type of the class from which it is being deleted. This advocates that we should reinsert all other paths that will not, in an inferred manner, maintain the removed instantiation. When we have deleted the necessary class instantiations and reinserted the irrelevant ones, we check to see which properties are no longer valid. As with classes, property instances may have inferred super-property instances. It may occur that the reclassification of a resource as a result of a delete operation may invalidate a property instance but its superproperty may still be valid. Therefore we opt to reclassify the property instance in an analogous manner if it is possible.

Property Instance Insertion Insertion of a property instance under a given graphset is similar process to that of inserting a class instance. The first step is to find and delete all property instantiations that will no longer need to be stored as they will be implied by the new property instantiation through RDFS inference. Subsequently, we actually introduce the new instantiation link. Unlike classes however, inserting property instances requires some additional conditions to be met. To insert a prop-

erty instance the subject and object must be existing resources. Moreover, their type must be of equal or descendant type to that of the domain and range of the property respectively. Should these conditions not be met the insertion fails and no change is made.

Property Instance Deletion The deletion of a property instances is again similar to that of class instances. Though there are no validity sideffects to worry about, the goal to minimize changes to those specified remains. First we find all property instance links that are explicitly stored and that, together with the existing schema information, infer the triple to be deleted. These triples must have the same subject and same object as the instance being deleted. These are the the triples that are actually going to be deleted because, as long as they exist, their inferred triples will also exist. Just as with classes, in the case of DAGs it is not necessary to remove the entire instantiated graph, but only the path that instantiates the property being deleted. Therefore we must reinsert all other paths that will not, in an inferred manner, maintain the removed instantiation.

Chapter 8

Related Work

In this chapter we will compare our extention of RUL with related languages capable of querying or updating named graphs.

8.1 SPARQL

SPARQL (SPARQL Protocol and RDF Query Language) is a protocol [21] and RDF query language [28], which as of January 2008 has become an official W3C Recommendation. Syntactically, it borrows characteristics from basic SQL and the Turtle RDF syntax [2]. It uses Graph patterns as conjunctions of RDF triples in the `WHERE` clause. Variables are used for selection, joins, and complex conditions. The variables to be included in the result are declared in the `SELECT` clause. Alternatively to the latter the `CONSTRUCT` clause can be used to specify a graph pattern to be instantiated against variable bindings. The `FROM` clause specifies the URL(s) of the data graph(s) to be queried. The `FILTER` clause is used to express conditions over query variables. Finally, the `OPTIONAL` clause allows for the declaration of variables that need not exist for the result to be returned (outer join).

SPARQL also has built-in support for named graphs. These are mainly used as a scoping mechanism. Use of the `GRAPH` keyword in the `WHERE` clause allows the restriction of the results to those triples originating from a specific named graph. Graphs can appear in subject and object variaples of triple patterns. Here is an example of a SPARQL query that uses named graphs to constrain the origin of the results.

```
PREFIX foaf:  <http://xmlns.com/foaf/0.1/>
PREFIX data:  <http://example.org/foaf/>
SELECT ?nick
FROM NAMED <http://example.org/foaf/bobFoaf>
WHERE
```

```
   {
      GRAPH data:bobFoaf {
         ?x foaf:mbox <mailto:bob@work.example> .
         ?x foaf:nick ?nick }
}
```

In this example we want to find, based only on bobs Foaf named graph the nick of the person with mail ¡mailto:bob@work.example¿.

To some degree, SPARQL ows its simplicity to its lack of features such as reasoning. The language operates at a lower triple level and is agnostic to any RDFS schema. Only non-recursive queries can be therefore supported by SPARQL.

Many RDF platforms implementing SPARQL query engines such as Jena [17] also allow for *reasoners* to be applied on RDF before storing them . For example, an RDFS-reasoner will add to an RDF model all triples that can be inferred using RDFS inference rules (see section 3.2.3). However, these reasoners are agnostic to the RDF construct of named graphs and thus do not support reasoning over triples that span more than one named graph. As a result these platform cannot support provenance tracking when provenance information also involves schema triples.

8.2 SPARQL/Update

SPARQL/Update is a declarative language for expressing updates on RDF graphs, which accompanies the SPARQL specication Although still in its early stages, it is intended to be a standard language for describing and exchanging RDF updates over the Semantic Web. SPARQL/Update is a companion language to SPARQL and is envisaged to be used in conjunction with SPARQL query language. SPARQL/Update is not a W3C recommendation like SPARQL. It supports both atomic and bulk updates to RDF graphs. It also offers the insertion and deletion of entire RDF named graphs. Here is an update example:

```
PREFIX foaf:  <http://xmlns.com/foaf/0.1/>
PREFIX data:  <http://example.org/foaf/>
PREFIX xsd:  <http://www.w3.org/2001/XMLSchem#>

   INSERT INTO <http://example.org/foaf/bobFoaf>
   { <http://example.org/foaf/bob> foaf:knows ?friend }
WHERE
   { GRAPH <http://example.org/foaf/aliceFoaf>
```

```
      <http://example.org/foaf/alice> foaf:knows ?friend .
      }
   }
```

In this update we add all friends of alice found in her named graph as friends of bob in his graph.

```
DELETE FROM <http://example.org/foaf/bobFoaf>
   { <http://example.org/foaf/bob> foaf:knows ?friend }
WHERE
   { GRAPH <http://example.org/foaf/aliceFoaf>
      <http://example.org/foaf/alice> foaf:knows ?friend .
      }
   }
```

Here we perform the oposite update. We remove all friends of alice found in her named graph from friends in bob's graph.

SPARQL/Update does not consider a named graph as an instantiation of a class and therefore does not introduce a new `typeof` triple when a new named graph is created. Moreover, a named graph can only be created in a direct fashion using an explicit URI, in other words, named graph cannot be created using variables from triple patterns. Finally, SPARQL doesn't allow the creation of a named graph if the URI already exists. This disables the classification of a resource as a named graph and as some other type. This is the syntax used by SPARQL/Update to create a new named graph:

```
CREATE GRAPH <uri>
```

8.3 TriQL

Among the first query languages created to query named graphs was TriQL [4]. Is is a simple graph pattern based query language where each pattern consists of a set of triple patterns and an optional graph named. However with the expressive power of SPARQL that followed TriQL replaced the latter.

8.4 TRIPLE

The early TRIPLE language [37] is a layered and modular rule language for the semantic web, based on Horn logic. It borrows many basic features from F-Logic but is especially designed for querying and transforming RDF models. It's architecture allows for modular

logic extentions such as RDFS. Moreover it provides built in support for contextualisation which is one of the central aims of the named graph extention to RDF.

8.5 Other Languages

There are other approaches to querying RDF ontologies appart from relational query languages. Xcerpt [33] is a deductive, rule-based query language for graph-structured data. It features high expressivenes and power followed by a complexity and scalability issues. Algae [29] is a reactive rule query language. Unlike SPARQL, both these languages inherently support any kind of inferrence through the use of rules. However they do not currently support named graphs.

8.6 Named Graph Extentions

This work has been based on the concept of RDF named graphs. Originally named graphs were introduced in [9]. However we have deviated from the original semantics of named graphs so that they may be usefully incorporated in an RDF query and update language. Firstly, we dissalow the instantiation of named graphs in other named graphs or themselves. This is primarily because this is not supported by existing serializations such as TriX or TriG. It also persents semantically complicated situations when deleting named graphs that contain the instantiations of other named graphs. Most importantly however, we have extended the notion of named graphs to incorporate RDFS schema semantics. Other works have provided different mechanisms for similar problems. [35] proposed Networked Graphs, an extention over named graphs that, apart from the direct association of triples with a graph enables the definition of graphs as views over an existing graphs. These views are expressed as SPARQL queries. As a result this work is also agnostic to any type of reasoning.

Chapter 9

Conclusions & Future Work

In recent years there has been a significant increase in science based innovation which in turn provokes the need for new conceptual and technological tools from computer science. Following the apparent shift from traditional science to e-science, databases are playing an ever increasing role storing raw data or results from computational analysis, experiments, simulations as well as annotations on data. Scientific applications have very different requirements from databases in comparison to traditional business applications. They require association of Semantics with data such as data quality or provenance, as well as the ability to query and update the data by people who are not necessarily database experts. In this work we studied the problem of provenance tracking on currated collaborative scientific databases built with RDF/S. Currated databases contain data that are usually collected from many sources and are constantly and often manually populated and corrected.

Many attempts are being made to integrate scientific data in fields such as biology and chemistry into Semantic Web ontologies[42][31]. One of the key requirements of such collaborative attempts is the ability to track the origin of each piece of information i.e. its provenance. Support for provenance in RDF was achieved with the introduction of Named Graphs [9], a construct allowing the reification of RDF graphs. However, Named Graphs are oblivious to schema information. In the presence of class and property hierarchies a lot of information is not explicit but inferred from various other bits of information that can originate from many different sources. In this thesis we presented a theoretical extention over RDF/S and Named Graphs that can capture provenance of information originating from multiple sources. We achieved this through a new construct, the *Graphset*.

Also in this work we formalized the semantics of querying and updating ontologies that use graphsets to capture provenance. Moreover, we extended the RUQL query and update language to support these semantics. Finally, we implemented these changes on

the RUQL interpreter part or RDF-Suite[1].

As with Named Graphs, graphsets are not restricted to describing provenance information. They can be potentially utilized to enable functionalities such as ontology evolution (versioning) or access control, among others.

The problem of integrating heterogenous scientific data from many different sources has indeed countless challenges yet to be tackled. Our focus was on giving a solution to the problem of tracking information provenance in collaborative RDF/S ontologies that are the result of such data integration. In the future we aim to extend the semantics of our update mechanisms to support updates on schema information. Moreover, there is work to be done in improving the complexity of the algorithms involved in calculating the graphsets to which inferred information belongs to. Also, a future goal would be to define the semantics of the replace operation for RDF/S graphs in the presence of graphsets. Finally, we hope to extend our formalisation to capture how a piece of information has originated from its sources. In other words in what way have the individual bits of data contributed to the inference of a particular piece of information.

Bibliography

[1] So Alexaki, Vassilis Christophides, Greg Karvounarakis, Dimitris Plexousakis, and Karsten Tolle. The ics-forth rdfsuite: Managing voluminous rdf description bases. pages 1–13, 2001.

[2] David Beckett. Turtle - Terse RDF Triple Language. `jena.hpl.hp.com/~afs/SPARQL-Update.html`.

[3] T. Berners-Lee, T. Hendler, and J. Lassila. The Semantic Web. SCIENTIFIC AMERICAN-AMERICAN EDITION-, 2001.

[4] Cris Bizer. TriQL - A Query Language for Named Graphs. `www4.wiwiss.fu-berlin.de/bizer/TriQL/`, April 2004.

[5] T. Bray, D. Hollander, A. Layman, and R. Tobin. Namespaces in XML 1.0 (Second Edition). `www.w3.org/TR/REC-xml-names/`, August 2006. W3C Recommendation.

[6] D. Brickley and R.V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. `www.w3.org/TR/2004/REC-rdf-schema-20040210/`, February 2004.

[7] Jeen Broekstra and Arjohn Kampman. Serql: A second generation rdf query language. In *In Proc. SWAD-Europe Workshop on Semantic Web Storage and Retrieval*, pages 13–14, 2003.

[8] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Tractable Reasoning and Efficient Query Answering in Description Logics: The DL-Lite Family. *Journal of Automated Reasoning*, 39(3), 2007.

[9] J. Carroll, C. Bizer, P. Hayes, and P. Stickler. Named graphs, Provenance and Trust. In *WWW*, 2005.

[10] R.G.G. Cattell, D.K. Barry, M. Berler, J. Eastman, D. Jordan, C. Russell, C. Schadow, T. Stanienda, and F. Velez. *The Object Database Standard: ODMG 3.0*. Morgan Kaufmann Publishers, 2000.

[11] V. Christophides, D. Plexousakis, M. Scholl, and S. Tourtounis. On labeling schemes for the semantic web. In *WWW*, 2003.

[12] B. McBride F. Manola, E. Miller. RDF Primer. `www.w3.org/TR/rdf-primer/`, February 2004.

[13] P. Gardenfors. Belief Revision: An Introduction. *Belief Revision*, (29):1–28, 1992.

[14] C. Gutierrez, C. A. Hurtado, and A. O. Mendelzon. Foundations of Semantic Web Databases. In *PODS*, 2004.

[15] P. Hayes. RDF Semantics. `www.w3.org/TR/rdf-mt/`, February 2004. W3C Recommendation.

[16] International Union of Basic and Clinical Pharmacology. `www.iuphar.org`.

[17] Jena – A Semantic Web Framework for Java. `jena.sourceforge.net/`.

[18] G. Karvounarakis. Querying RDF metadata for Community Web Portals. Master's thesis, 2002.

[19] G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl. Rql: a declarative query language for rdf. pages 592–603. ACM Press, 2002.

[20] G. Karvounarakis, A. Magkanaraki, S. Alexaki, V. Christophides, D. Plexousakis, M. Scholl, and K. Tolle. *Functional Approach to Data Management: Modelling, Analyzing and Integrating Heterogeneous Data*, chapter RQL: A Functional Query Language for RDF. LNCS Series, Springer-Verlag, 2004.

[21] Elias Torres Kendall Grant Clark, Lee Feigenbaum. SPARQL Protocol for RDF. `www.w3.org/TR/rdf-sparql-protocol/`, January 2008.

[22] G. Konstantinidis, G. Flouris, G. Antoniou, and V. Christophides. A Formal Approach for RDF/S Ontology Evolution. In *ECAI*, 2008.

[23] M. Magiridou, S. Sahtouris, V. Christophides, and M. Koubarakis. RUL: A Declarative Update Language for RDF. In *ISWC*, 2005.

[24] S. Munoz, J. Perez, and C. Gutierrez. Minimal deductive systems for RDF. In *ESWC*, 2007.

[25] A. Newman, J. Hunter, Y. F. Li, C. Bouton, and M. Davis. A Scale-Out RDF Molecule Store for Distributed Processing of Biomedical Data.

[26] Pellet. `pellet.owldl.com`.

[27] J. Perez, M. Arenas, and C. Gutierrez. nSPARQL: A Navigational Language for RDF. In *ISWC*, 2008. To appear.

[28] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. `www.w3.org/TR/rdf-sparql-query/`, January 2008.

[29] Eric Prud'hommeaux. Algae RDF Query Language. `www.w3.org/2004/05/06-Algae/`, 2004.

[30] Racer. `www.racer-systems.com`.

[31] BioPAX : Biological Pathways Exchange. `www.biopax.org`.

[32] A. Ruttenberg, T. Clark, W. Bug, M. Samwald, O. Bodenreider, H. Chen, D. Doherty, K. Forsberg, and H. Cheung. BMC Bioinformatics Methodology Advancing translational research with the Semantic Web.

[33] Sebastian Schaffert and François Bry. Querying the web reconsidered: A practical introduction to xcerpt. In *In Proc. Extreme Markup Languages*, 2004.

[34] S. Schenk and S. Staab. Networked graphs: a declarative mechanism for SPARQL rules, SPARQL views and RDF data integration on the Web. In *WWW*, 2008.

[35] Simon Schenk and Steffen Staab. Networked rdf graphs. Technical report, 2007.

[36] G. Serfiotis, I. Koffina, V. Christophides, and V. Tannen. Containment and Minimization of RDF/S Query Patterns. In *ISWC*, 2005.

[37] M. Sintek, D. Gmbh, and S. Decker. Triple—an rdf query, inference, and transformation language. In *In Proc. Deductive Database and Knowledge Management*, 2001.

[38] Michael Sintek, Dfki Gmbh, and Stefan Decker. Triple—an rdf query, inference, and transformation language. In *In Proc. Deductive Database and Knowledge Management*, 2001.

[39] Y. Theoharis, V. Christophides, and G. Karvounarakis. Benchmarking Database Representations of RDF/S Stores. In *4th Int'l Semantic Web Conference*, November 2005.

[40] URI Planning Interest Group, W3C/IETF. URIs, URLs, and URNs: Clarifications and Recommendations 1.0. `www.w3.org/TR/uri-clarification/`, September 2001.

[41] W3C. The Semantic Web layer cake. www.w3.org.

[42] Semantic Web Health Care and Life Sciences (HCLS) Interest Group. www.w3.org/2001/sw/hcls/.

Appendix A

The BNF grammar of RUL

ns_query ::= (*query* | *strict_query*) ["`using namespace`" *nsdeflist*] ["`using namedgraph`" *ngdeflist*
ns_update ::= *update* ["`using namespace`" *nsdeflist*] ["`using namedgraph`" *ngdeflist*]
update ::= *rul_statement* { *rul_statement* } ["`from`" *rangeslist*]
| *rul_statement* ["`from`" *rangeslist* "`where`" *query*]
rul_statement ::= (*ins_del_statement* | *replace_statement*)
ins_del_statement ::= ("`delete`" | "`insert`") *ins_del_expression* { "`,`" *ins_del_expression* }
replace_statement ::= ("`replace`") *replace_expression* { "`,`" *replace_expression* }
ins_del_expression ::= [*identifier::*](*prefixed_id* | *class_var*) "`(`" *res_expr* "`)`"
| [*identifier::*] "`&`" *identifier*
| [*identifier::*] (*prefixed_id* | *property_var*) "`(`" *object* "`,`" *object* "`)`"
replace_expression ::= (*prefixed_id* | *class_var*) "`(`" *res_expr* "`<-`" *res_expr* "`)`"
| (*prefixed_id* | *class_var*) "`<-`" (*identifier* | *class_var*) "`(`" *res_expr* "`)`"
| *identifier* "`<-`" *identifier*
| (*prefixed_id* | *property_var*) "`(`" *res_expr* "`<-`" *res_expr* "`,`" *object* "`)`"
| (*prefixed_id* | *property_var*) "`(`" *res_expr* "`,`" *object* "`<-`" *object* "`)`"
| (*prefixed_id* | *property_var*) "`(`" *res_expr* "`<-`" *res_expr* "`,`" *object* "`<-`" *object* "`)`"
| (*prefixed_id* | *property_var*) "`<-`" (*identifier* | *property_var*) "`(`" *res_expr* "`,`" *object* "`)`"
res_expr ::= (*data_var* | "`&`" *identifier*)
object ::= (*data_var* | *constant*)
query ::= "`(`" *query* "`)`"
| "`subClassOf`" ["`^`"] "`(`" *query* "`)`"
| "`superClassOf`" ["`^`"] "`(`" *query* "`)`"
| "`subPropertyOf`" ["`^`"] "`(`" *query* "`)`"
| "`superPropertyOf`" ["`^`"] "`(`" *query* "`)`"
| "`topclass`"
| "`topproperty`"
| "`leafclass`" ["`(`" *query* "`)`"]
| "`leafproperty`" ["`(`" *query* "`)`"]

| "nca" "(" *query* "," *query* ")"
| "domain" "(" *query* ")"
| "range" "(" *query* ")"
| "typeOf" "(" *query* ")"
| "namespace" "(" *query* ")"
| "count" "(" (*query* | *strict_query*) ")"
| "avg" "(" (*query* | *strict_query*) ")"
| "min" "(" (*query* | *strict_query*) ")"
| "max" "(" (*query* | *strict_query*) ")"
| "sum" "(" (*query* | *strict_query*) ")"
| "bag(" *query, [query]*")"
| "seq(" *query, [query]*")"
| "gset(" *query, [query]*")"
| *query* "[" *query* "]"
| *query* "in" (*query* | *strict_query*)
| *(query* | *strict_query) set_op (query* | *strict_query)*
| *query comp_op query*
| *query bool_op query*
| "not" *query*
| *constant*
| prefixed_id
| *var*
| *sfw_query*
| "exists" *var* "in" *(query* | *strict_query)* "such that" *query*
| "forall" *var* "in" *(query* | *strict_query)* "such that" *query*

strict_query ::= "^" prefixed_id
| "^" *var*
| "(" *strict_query* ")"

sfw_query ::= "select" *[* "distinct" *] projslist* "from" *rangeslist [* "where" *query]*

comp_op ::= "<" | "<=" | ">" | ">=" | "=" | "!=" | "like"

set_op ::= "union" | "intersect" | "minus"

bool_op ::= "and" | "or"

constant ::= integer_literal
| real_literal
| quoted_string_literal
| quoted_char_literal
| date
| "true"
| "false"
| "&" identifier

var ::= *data_var* | *class_var* | *type_var* | *property_var*

data_var ::= identifier

class_var ::= "$" identifier

type_var ::= "$" "$" identifier

property_var ::= "@" identifier

projslist ::= "*" | *query* { "," *query* }

rangeslist ::= *pathexpr* { "," *pathexpr* }

pathexpr ::= *[identifier::]* *pathelem* { "." *pathexpr* }

pathelem ::= *[* "{" *from_to* "}" *]* (*query* | *strict_query*) *[* "{" *from_to* "}" *]*
| (*query* | *strict_query*) ";" *property_var*

from_to ::= *[data_var] [* ";" *[* "^" *]* (*class_var* | *type_var* | identifier) *]* | *class_var* | *type_var*

nsdeflist ::= *nsdef* { "," *nsdef* }

nsdef ::= identifier "=" "&" identifier

ngdeflist ::= *ngdef* { "," *ngdef* }

ngdef ::= identifier "=" "&" identifier
| identifier "=" *[* "^" *]* "(" "&" identifier { "," "&" identifier } ")"

prefixed_id ::= *[identifier* ":" *]* *identifier*