University of Crete

Computer Science Department

Optimising and Reformulating RQL Queries on the Semantic Web

Georgios Serfiotis

Master's Thesis

Heraklion, March 2005

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

ΤΜΗΜΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

Βελτιστοποιώντας και Αναδιατυπώνοντας RQL Επερωτήσεις στο Σημασιολογικό Ιστό

Εργασία που υποβλήθηκε από τον

Γεώργιο Σερφιώτη

ως μερική εκπλήρωση των απαιτήσεων για την απόκτηση

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΕΙΔΙΚΕΥΣΗΣ

Συγγραφέας:

____

Γεώργιος Σερφιώτης, Τμήμα Επιστήμης Υπολογιστών

Εισηγητική Επιτροπή:

____

Βασίλης Χριστοφίδης, Αναπληρωτής Καθηγητής, Επόπτης

____

Γρηγόρης Αντωνίου, Καθηγητής, Μέλος

____

Αναστασία Αναλυτή, Ερευνήτρια Ινστιτούτου Πληροφορικής ΙΤΕ, Μέλος

Δεκτή:

____

Δημήτρης Πλεξουσάκης, Αναπληρωτής Καθηγητής

Πρόεδρος Επιτροπής Μεταπτυχιακών Σπουδών

Ηράκλειο, Μάρτιος 2005

Optimising and Reformulating RQL Queries on the Semantic Web

Georgios Serfiotis

Master's Thesis

Computer Science Department, University of Crete

Abstract

A cornerstone issue in the realisation of the Semantic Web (SW) vision is the achievement of semantic interoperability among legacy data sources spread worldwide. In order to capture information semantics in a machine processable way, various ontology-based formalisms have been recently proposed (e.g., RDF/S). However, the vast majority of existing legacy data is not yet in RDF/S or any other SW language. As a matter of fact, most of the data is physically stored in relational database (RDB) systems and published on the Web as XML.

SW applications, however, require to view data as virtual RDF, valid instance of a domain or application specific RDF/S schema, and to be able to manipulate them with high-level query languages, such as RQL. Therefore, we propose a middleware system that allows querying RDB data using RQL.

So, our work focuses on specifying a first-order logic encoding for RDF, namely SWLF, along with constraints preserving RDF/S semantics that will allow specifying RDB to RDF mappings, composing RQL queries with these mappings - thus, producing RDB queries (a.k.a query reformulation) - and performing semantic query optimisations.

In particular, we focus on RQL query containment and minimisation. By employing minimisation (optimisation) techniques we may reduce the requirements in time and space, which are two very valuable resources when managing queries, especially over distributed systems, in order to answer a query. The optimisation removes RQL query redundancy (by taking advantage of the RDF/S constraints) and redundancy of the reformulated query (by exploiting constraints of the underlying RDB and other).

Supervisor: Vassilis Christophides
Associate Professor

Βελτιστοποιώντας και Αναδιατυπώνοντας RQL Επερωτήσεις στο Σημασιολογικό Ιστό

Γεώργιος Σερφιώτης

Μεταπτυχιακή Εργασία

Τμήμα Επιστήμης Υπολογιστών, Πανεπιστήμιο Κρήτης

Περίληψη

Ο θεμέλιος λίθος για την πραγματοποίηση του οράματος του Σημασιολογικού Ιστού είναι η επίτευξη της σημασιολογικής διαλειτουργικότητας μεταξύ υπαρχόντων πηγών δεδομένων σε οργανισμούς ανά τον κόσμο. Με στόχο την περιγραφή της σημασιολογίας των πληροφοριών με ένα μηχανικά αντιληπτό τρόπο, διάφοροι φορμαλισμοί βασισμένοι σε οντολογίες έχουν πρόσφατα προταθεί (π.χ., RDF/S). Παρόλα αυτά, η μεγάλη πλειοψηφία των υπαρχόντων δεδομένων δεν είναι ακόμα σε μορφή RDF/S ή άλλης γλώσσας του Σημασιολογικού Ιστού. Για την ακρίβεια, τα περισσότερα δεδομένα είναι αποθηκευμένα σε σχεσιακά συστήματα βάσεων δεδομένων (RDB) και δημοσιευμένα στον Παγκόσμιο Ιστό ως XML.

Παρόλα αυτά, οι εφαρμογές του Σημασιολογικού Ιστού απαιτούν να βλέπουν τα δεδομένα ως εικονική RDF, έγκυρο στιγμιότυπο ενός RDF/S σχήματος καθορισμένου πεδίου ή εφαρμογής, και να μπορούν να τα χειρίζονται με υψηλού επιπέδου γλώσσες επερώτησης, όπως η RQL. Επομένως, προτείνουμε ένα σύστημα διαμεσολάβησης το οποίο επιτρέπει την επερώτηση σχεσιακών δεδομένων χρησιμοποιώντας RQL.

Γι’ αυτό, η δουλειά μας εστιάζεται στον ορισμό μίας λογικής κωδικοποίησης πρώτης τάξης για την RDF – την SWLF – μαζί με περιορισμούς που διατηρούν τη σημασιολογία της RDF/S. Τα παραπάνω θα επιτρέψουν τον καθορισμό αντιστοιχήσεων μεταξύ του RDF/S και του σχεσιακού σχήματος, τη σύνθεση των RQL επερωτήσεων με αυτές τις αντιστοιχήσεις – παράγοντας σχεσιακές επερωτήσεις (αναδιατύπωση επερωτήσεων) – και τη βελτιστοποιήση επερωτήσεων.

Ιδιαίτερη έμφαση δίνεται στον εγκλεισμό και στην ελαχιστοποίηση των RQL επερωτήσεων. Χρησιμοποιώντας τεχνικές ελαχιστοποίησης (βελτιστοποίησης) μπορούμε να μειώσουμε τις απαιτήσεις για την απάντηση επερωτήσεων σε χρόνο και χώρο, οι οποίοι είναι πολύτιμοι πόροι, ιδιαίτερα πάνω από κατανεμημένα συστήματα. Η βελτιστοποίηση εξαλείφει τους πλεονασμούς τόσο από τις RQL επερωτήσεις (εκμεταλλευόμενη τους RDF/S περιορισμούς) όσο και από τις αναδιατυπωμένες επερωτήσεις (εκμεταλλευόμενη τους περιορισμούς τις σχεσιακής βάσης δεδομένων).

Επόπτης Καθηγητής: Βασίλης Χριστοφίδης
Αναπληρωτής Καθηγητής

Στους γονείς μου και
στον αδελφό μου Ανδρέα

Ευχαριστίες

Στο σημείο αυτό θα ήθελα να ευχαριστήσω τον επόπτη μου κ. Βασίλη Χριστοφίδη για τα όσα μου προσέφερε στα δυόμιση και πλέον χρόνια της συνεργασίας μας. Η βοήθεια του ήταν πολύτιμη και η καθοδήγησή του καθοριστική για την ολοκλήρωση της εργασίας αυτής. Ιδιαίτερα θέλω να τον ευχαριστήσω για τη συμπαράσταση του στη δύσκολη περίοδο προ της ολοκλήρωσής της. Ελπίζω να φάνηκα αντάξιος των προσδοκιών του.

Θα ήθελα να ευχαριστήσω και την κα. Αναστασία Αναλυτή και τον κ. Γρηγόρη Αντωνίου, τα δύο άλλα μέλη της επιτροπής εξέτασης της μεταπτυχιακής μου εργασίας, για τις πολύτιμες παρατηρήσεις τους.

Επίσης, θα ήθελα να ευχαριστήσω το Πανεπιστήμιο Κρήτης και την ομάδα Πληροφοριακών Συστημάτων του Ινστιτούτου Πληροφορικής για τα όσα μου προσέφεραν και όσα αποκόμισα κατά τις σπουδές στην Κρήτη.

I would like to thank Professor Val Tannen for his contribution in understanding the idea behind this thesis. Additionally, I would like to thank Alin Deutsch and Nicola Onose, since by experimenting with their system we had many questions and they were more than willing to answer them.

Ακόμη, θα ήθελα να ευχαριστήσω τους φίλους μου, ιδιαίτερα αυτούς των φοιτητικών μου χρόνων (Βάσω, Γιώργο, Δέσποινα, Ιωάννα, Νίκο και Παναγιώτη) για όσα περάσαμε μαζί και τη συμπαράστασή τους στις δύσκολες στιγμές. Ιδιαίτερα θα ήθελα να ευχαριστήσω την Ιωάννα για την επί δύομιση χρόνια στενή συνεργασία μας και για την ανοχή της στις αργοπορημένες μου αφίξεις!

Το μεγαλύτερο, όμως, ευχαριστώ το αξίζουν οι γονείς μου, Διαμαντής και Νέλλα, και ο αδελφός μου Ανδρέας για την αμέριστη συμπαράστασή τους και την εμπιστοσύνη που μου έδειξαν όλα αυτά τα χρόνια. Σας ευχαριστώ πολύ για όλα.

Γιώργος Σερφιώτης

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Integration is one of the most pressing and expensive problems faced today by companies and organisations maintaining a multitude of legacy databases – usually relational databases – and corresponding applications. These systems usually contain valuable information and are often still good for supporting specific tasks. Unfortunately, the information they contain cannot be leveraged by other systems without considerable effort.

Until recently, the most common solution to integration was the field to field mapping, where schemas from two data sources are imported and fields are mapped to each other. However, this solution has many drawbacks. First of all, it presents scalability problems, because the number of mappings increases exponentially with the number of sources. Moreover, the maintenance and evolution of the mappings is very demanding; each change in one source reflects in all corresponding mappings. Furthermore, the definition of the mappings presupposes that the person responsible is familiar with both sources, which usually is not the case.

A first attempt to overcome these problems was made by moving towards XML, in order to take advantage of a universal data encoding when integrating legacy systems in the Web. However, XML does not capture the contextual meaning (semantics) of the data. Therefore, organisations and companies have started moving towards semantic technologies; data sources get mapped to domain ontologies, which, defined using ontology languages, describe the concepts of the domain and the relationships between them, i.e. the domain's semantics. Ontologies enable communication between computer systems in a way that is independent of the individual system technologies, information architectures and applications. Moreover, by adopting ontology languages that describe semantics in a machine processable way – like RDF/S ([MM04]) and OWL ([HM04]), legacy systems can get integrated in the Semantic Web ([SW01]).

There exist two approaches to semantic integration: data warehousing and on-demand integration. In data warehousing all data of the legacy source get translated into data of the language describing the ontology as soon as the mapping procedure is completed. However, if we take under account the size of legacy databases in companies and organisations, this approach is often very expensive. Moreover, it demands constant synchronisation of the data produced with the database. On the contrary, on-demand integration can prove very useful, because the translation of the whole database is not needed. Each time a query is posed on the global (domain) ontology, data are collected from the integrated sources dynamically and outputted as virtual data of the ontology language.

On-demand integration presupposes a SW integration middleware (SWIM) that will allow users to: *(a)* specify correspondences (mappings) between RDB (and XML, since a lot of relational sources publish their data as virtual XML) sources and mediated RDF/S schemas, *(b)* verify that these mappings conform to the semantics of the employed schemas, *(c)* reformulate RQL queries against the underlying relational or XML sources using the mappings, *(d)* employ in the queries RVL views, *(e)* check queries for containment, and *(f)* perform query optimisations.

Lack of background work on query optimisation and reformulation for RQL or any other RDF/S query language, the challenge is to find a way to reduce the above problems into equivalent ones that can exploit already existent knowledge. The current thesis tries to reveal certain aspects of a SWIM, as well as the necessary decisions taken in order to make it feasible.

1.1 Motivating Examples

Suppose that there exists a relational database holding information about artefacts, like their title, creator and exhibition place (bottom part of Figure 1-1). Normally this data can be queried using SQL. But now, assume an RDF/S cultural schema, part of which is shown at the top of Figure 1-1. Then, RQL and RVL can be used to query this mediated schema and define views over it. For example, the query

```
SELECT    Y
FROM      {X}creates{Y}
```

returns the URIs of artefact (including painting and sculpture) resources created by some artist (perhaps painter or sculptor).

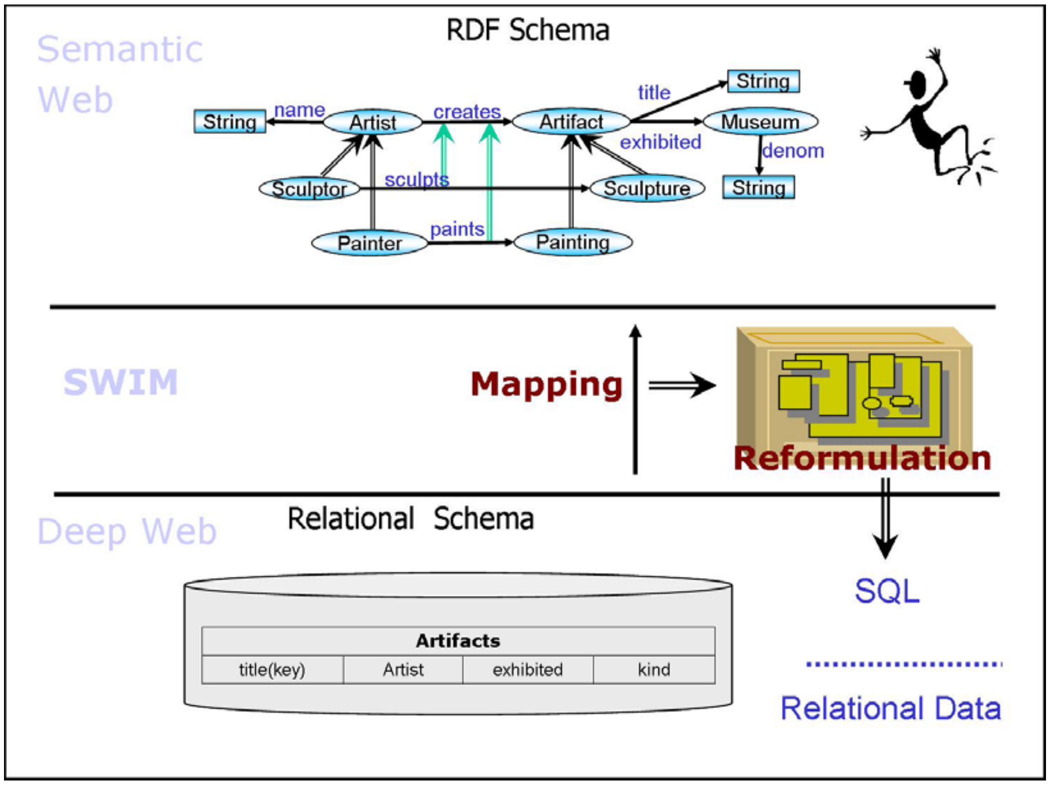


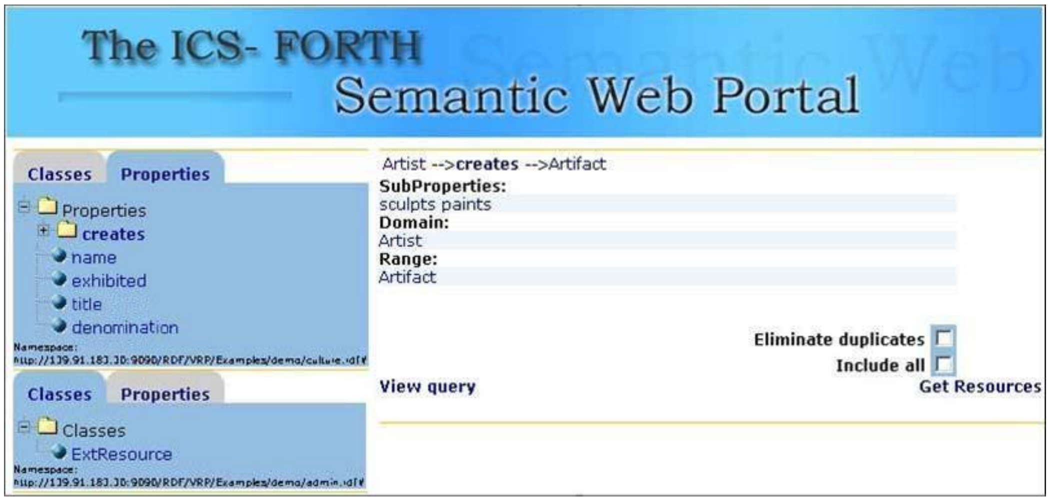
Figure 1-1: Mediation scenario: Publishing RDB as RDF

However, this RQL query cannot be answered directly, since there are no actual data; the RDF/S layer is virtual. Therefore, a middleware is needed that will reformulate the RQL query into an SQL one based on the relationships between the relational and the RDF/S schema. A formal way to express such relationships is the use of mappings from RDB to RDF. Such a reformulation procedure could rewrite the RQL query to the following SQL query

```
SELECT      a.Title
FROM        Artifacts a
WHERE       a.Kind="Painting"

UNION

SELECT      a.Title
FROM        Artifacts a
WHERE       a.Kind="Sculpture"
```



Namespace:
http://139.91.183.30:9090/RDF/VRP/Examples/demo/culture.rdf

Artist -->**creates** -->Artifact
SubProperties:
sculpts paints
Domain:
Artist
Range:
Artifact

Eliminate duplicates ☐
Include all ☐

Namespace:
http://139.91.183.30:9090/RDF/VRP/Examples/demo/admin.rdf

Figure 1-2: On the fly creation of RQL query

A similar case is the reformulation of RQL queries to XML queries when a virtual RDF/S schema is positioned on top of an XML repository.

Additional functionality like semantic optimisation of the RQL queries can prove very profitable in several cases and should be incorporated in such a middleware. Although RQL queries written by humans rarely contain any redundancy, this is not the case with machine-generated queries. Take for example the graphical RQL interface presented in [ACK04] that generates on the fly queries for the Semantic Web. Such a tool can be used for creating RQL queries through navigating on an RDF/S schema, virtual or not.

Look at Figure 1-2. While navigating through the properties of the RDF/S schema of Figure 1-1, we select property “creates”. This choice generates the query

```
SELECT    X, Y
FROM      {X}creates{Y}
```

that returns the extent of property “creates”, i.e. all artists and the artefacts they have created. If for some reason we decide to refine our query by selecting the property “paints”, a new query returning both the extents of “creates” and “paints” is produced. However, this conjunction is redundant. It is obvious that

```
SELECT    X, Y
FROM      {X}paints{Y}, {X}creates{Y}
```

is equivalent to

```
SELECT    X, Y
FROM      {X}paints{Y}
```

Although this tool for graphically generating RQL queries captures this redundancy and minimises the query, there is a fragment of RQL it cannot handle. To be more accurate this tool generates and minimises queries that belong to the fragment of RQL where queries are built on extended interpretations of classes[1] (and/or properties) and ask exclusively for data information. Therefore, it does not handle queries containing proper interpretations, like

```
SELECT    X
FROM      {X}^paints{Y}, {X}creates{Y}
```

Additionally, even for the RQL fragment it handles, no theorem has been proved stating that the minimised queries are minimal, i.e. cannot get further minimised.

Moreover, with the widespread use of the Semantic Web more graphical tools are expected to appear. Some of them may not offer optimisation services. Therefore, a framework that will allow minimising declarative RDF/S queries, such as RQL, is welcome, too.

1.2 Introducing the Semantic Web Integration Middleware (SWIM)

The previous examples made obvious the need for a Semantic Web integration middleware (SWIM) that will facilitate users to evaluate queries against virtual RDF/S schemas and will offer them optimisation services.

The selection of a framework that will treat the above problems uniformly is crucial. The specification of the mappings along with the ability to exploit well-established techniques for query reformulation and optimisation leads to the adaptation of a logic framework based on Linear Datalog, which has a straightforward

[1] A class' (property's) proper interpretation refers to the direct instances of the class (property). On the contrary, its extended interpretation refers to the direct instances of itself and of its subclasses (subproperties).

correspondence to the relational theory. The goal is to reduce the RQL to SQL and RQL to XQuery (XPath) reformulation problems to relational equivalents and reuse existing methods and results on relational query minimisation and reformulation.

In order to make feasible such a reduction, a relational representation of the RDF/S model has been incorporated in SWIM, based on which the virtual RDF/S schemas get translated into Datalog facts and the RDB to RDF mappings into Datalog rules. Using this information along with a set of predefined constraints capturing the semantics of RDF/S, the containment, minimisation and reformulation problems can be solved. The algorithms used to solve the problems are the chase and backchase [Deu02], which guarantee that both the RQL queries and the reformulated SQL ones can be minimised.

The current thesis addresses the RDF to RDB aspects of SWIM. The XML aspects are the subject of [Kof05].

1.3 Organisation

In the previous sections we introduced the problems that will concern us throughout this thesis and gave some motivation for our concern and an overview of the middleware whose functionality depends on solving these problems. The rest of the thesis is organised as follows:

Chapter 2 introduces the typical RDF data model as well as the data model of RQL and RVL, RDF's querying and view definition languages, respectively. Special attention is given on their semantics.

Chapter 3 presents the internal logical framework adopted in the SWIM middleware. The first-order (relational) relations and constraints used to capture the RDF data model and semantics are analysed. Then, the differences between the RDF semantics and the semantics captured by the logical framework are presented. Finally, the translation of (unions of) conjunctive RQL queries, namely RQL_{UCQ}, and RDF/S schemas into the internal logical representation is illustrated.

Chapter 4 is the building block of this thesis. It defines the problems of RQL_{UCQ} containment and minimisation and presents the algorithms used to deal with them. These problems are translated to equivalent relational ones and solved using the chase and backchase algorithms. Additionally, a fragment of RQL_{UCQ}, namely RQL_{CORE}, is defined for which the above problems are usually solved easier.

Chapter 5 describes the whole reformulation process that starts with an incoming RQL_{UCQ} query and outputs equivalent reformulated SQL queries ready to be executed.

Chapter 6 presents SWIM’s architecture. One by one the components forming it are described and the choices made are justified. Moreover, relative systems and works are compared to SWIM.

Chapter 7 discusses some issues that deserve further investigation and, then, summarises this thesis.

Chapter 2

The Resource Description Framework (RDF)

RDF constitutes part of the activity coordinated by the World Wide Web Consortium (W3C). It is a general-purpose language for representing and exchanging descriptive information about *web resources* over the World Wide Web, e.g., *metadata* about those resources, like titles, dates, and authors of Web pages. The challenge is to enable the resource descriptions in a formal, interoperable, and humanly readable way via appropriate languages, without making any assumption about the application domain or the structure of the described information resources.

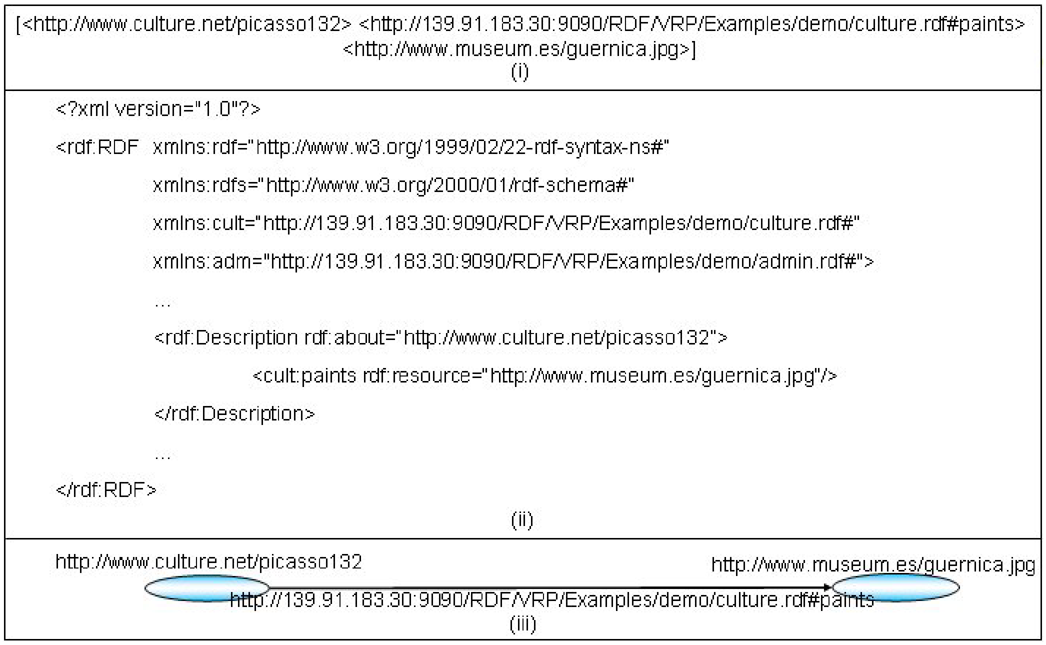
2.1 RDF: Model, Schema and Semantics

In RDF the concept "web resource" is generalised in order to capture anything that can be identified and not necessarily information resources that can be accessed on the Web. Every web resource is given a unique web identifier called *Universal Resource Identifier* or *URI*[2] ([CK04]) and gets described using simple *statements*. A statement consists of a specific resource along with a property and the property's value, called *subject*, *predicate* and *object*, respectively. For example, in a sentence stating that "Pablo Picasso painted Guernica", the URI referring to Pablo Picasso is the subject, the one referring to the property "painted" is the predicate and the value "Guernica" is the object. All statements about a specific resource form its *description*.

There are three ways to represent RDF statements ([MM04]): i) using *triples*, ii) using *directed labelled graphs* ([CK04]) and iii) using *XML syntax* ([Bec04]). In the triples notation each statement is written as a triple of subject, predicate and object, in that order. Alternatively, RDF statements can be modelled as *nodes* and *arcs* in a graph. According to this notation a statement is represented by one node for the subject, one for the object and an arc for the predicate directed from the subject to the

[2] See http://www.w3c.org/Addressing

object. Note that when using the graph model each resource corresponds to just one node; if it appears in more than one statement, all property arcs connect to the same node.



```
[<http://www.culture.net/picasso132> <http://139.91.183.30:9090/RDF/VRP/Examples/demo/culture.rdf#paints>
<http://www.museum.es/guernica.jpg>]
(i)
```

```
<?xml version="1.0"?>
<rdf:RDF  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
          xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
          xmlns:cult="http://139.91.183.30:9090/RDF/VRP/Examples/demo/culture.rdf#"
          xmlns:adm="http://139.91.183.30:9090/RDF/VRP/Examples/demo/admin.rdf#">
          ...
          <rdf:Description rdf:about="http://www.culture.net/picasso132">
                    <cult:paints rdf:resource="http://www.museum.es/guernica.jpg"/>
          </rdf:Description>
          ...
</rdf:RDF>
(ii)
```

http://www.culture.net/picasso132
http://www.museum.es/guernica.jpg
http://139.91.183.30:9090/RDF/VRP/Examples/demo/culture.rdf#paints
(iii)

Figure 2-1: A simple statement's representations

To exchange RDF statements in a machine-processable way the *Extensible Markup Language* ([BMPS00]), known under the acronym *XML*, is used; the specific XML dialect defined is called *RDF/XML*. The RDF/XML representation of statements is not necessarily unique; some statements can be modelled using more than one XML encodings. This is mainly due to the fact that properties can be modelled both as XML attributes and XML elements and to the fact that predicates may be nested or not. Figure 2-1 illustrates the three different representations for the example statement stated previously. Notice that in all the above representations the resources appear with their URI.

Each RDF statement consists, as previously stated, of a subject, a predicate and an object. Although, the subject is always a URI reference to the resource being described (drawn as an oval in the graph representation) and the predicate is always a URI reference to a property, the object is either another URI reference, case of Figure 2-1, or a literal (drawn as a rectangle in the graph representation), case of Figure 2-2.

Literals are simple strings either combined with a datatype URI (typed literals) or not (plain literals). Depending on the object's type, the predicate's correspondence is straightforward to a relation or an attribute.

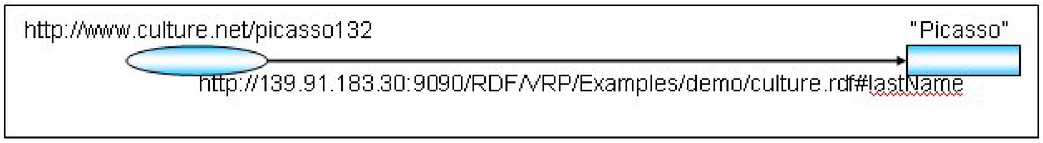

http://www.culture.net/picasso132
"Picasso"
http://139.91.183.30:9090/RDF/VRP/Examples/demo/culture.rdf#lastName

Figure 2-2: Statement example with literal as object

Moreover, people sometimes want to express statements about a collection of resources or literals. RDF supports three types of *containers* to make references to collections, namely *bags*, *sequences* and *alternatives*, where each container is itself an RDF resource. A bag represents a multi set of values, a sequence represents an ordered list of values and an alternative represents a group of resources (or literals) that are alternatives for a property's single value.

A very useful feature of RDF is the ability to use shorthands instead of full URI references in the triples representation. This way the space needed for writing the triples reduces significantly. Having in mind that a full URI reference is formed by a *URI namespace* and a *local name*, when a shorthand is used, a *prefix* is assigned to the URI namespace and the local name gets appended to it. The use of shorthands extends to the RDF/XML representation, too. Observe the example in Figure 2-1ii where four prefixes are introduced.

2.1.1 RDF Schema

Having seen how statements are formed and how they are represented, the next step is to find out how the vocabularies of terms employed by those statements are defined; i.e. how to describe the different classes of resources and the properties used to define resources and their values. RDF does not provide by default descriptions of application-specific classes (e.g. Painter) and properties (e.g. paints[3]); however, it provides the means needed to describe such classes and properties. These means form

[3] By convention in this thesis class names start with uppercase and property names with lowercase.

on their own an RDF vocabulary, i.e. a specialised set of predefined resources, referred to as *RDF Schema* ([BG04]). This vocabulary is found at "*http://www.w3c.org/2000/01/rdf-schema#*", which by convention is associated to prefix *rdfs*. In other words the RDF Schema provides a type system for RDF.

The basic notion found in RDF Schema is *class* that corresponds to the generic concept of *Type* or *Category*. The resources *rdfs:Class*, *rdfs:Resource* and the properties *rdf:type*, *rdfs:subClassOf* are used for describing classes:

- Every resource that has as value of the property rdf:type the resource rdfs:Class is a class according to RDF Schema.
- The property rfd:type is used to state that a resource is an instance of another resource.
- The property rdfs:subClassOf allows defining *class specialisations*; its meaning is that every instance of a class' specialisation is, also, an instance of the original class.

The other basic RDF element that allows describing and characterising classes is *property*. In RDF Schema properties are described using the RDF Schema class *rdf:Property* and the RDF Schema properties *rdfs:domain*, *rdfs:range* and *rdfs:subPropertyOf*.

- Every resource that has as value of the property rdf:type (is an instance of) the class rdf:Property is a property according to RDF Schema.
- The property rdfs:domain is used to indicate on which class' instances a specific property is applied.
- The property rdfs:range is used to indicate that the values of a particular property are either given by a typed literal or instances of a designated class.
- The property rdfs:subPropertyOf allows defining *property specialisations*; its meaning is that every instance of a property's specialisation is, also, an instance of the original property.

Another characteristic of RDF is that properties are defined independently of class definitions and have, by default, global scope (i.e. may apply to all classes), unless domain specifications are explicitly stated. Moreover, properties can have multiple domain and range definitions. However, they cannot have locally different ranges depending on their domains; any range applies to all domains of the property.

There is, also, a number of other RDF Schema built-in properties that can be used to provide documentation and other information about an RDF schema or about instances, like *rdfs:comment*, *rdfs:label*, *rdfs:seeAll*, *rdfs:isDefinedBy* and *rdfs:seeAlso*.

Generally speaking, the statements forming an RDF schema provide additional descriptive information about resources.

2.1.2 RDF Semantics

As discussed in the previous sections, RDF is intended to be used for expressing statements about resources in the form of a graph, using specific vocabularies (URIs of resources, names of properties, classes, etc.). In order to serve this purpose, the meaning of an RDF graph must be defined in a formal way that will allow determining with mathematical precision the conclusions that machines can draw from a given RDF graph. The *model theory* described in the *RDF Semantics* ([H04]) is used to define this formal meaning, i.e. specify the formal *semantics* of RDF/S[4]. The RDF Schema's semantic extensions to the RDF language are defined in the RDF Semantics, too.

A model theory assumes that the language refers to a *world* and describes the minimal conditions that a world must satisfy in order to assign an appropriate meaning for every expression in the language. A particular world is called an *interpretation*, thus a model theory can be better called "*interpretation theory*".

The exact definitions of the RDF and RDFS interpretations can be found in [Hay04]. Based on these interpretations several axiomatic rules are defined and several RDF axiomatic triples are considered. For example, the condition stating when a resource is a property, as seen in subsection 2.1.1, is an RDF axiomatic rule, while [<rdf:type> <rdf:type> <rdf:Property>] is an RDF axiomatic triple.

The definition of *class* and *property extension* is needed before proceeding with presenting some RDFS axiomatic rules. So, a class' extension is the set of things that are in the class and a property's extension is the set of object-value pairs that satisfy the property. The RDFS axiomatic rules state, between other things, that:

- The rdfs:subPropertyOf's extension is transitive and reflexive.

[4] Term used as an alternative for both RDF and RDF Schema.

- If the triplet [<x> <rdfs:subPropertyOf> <y>] exists, then the extension of *x* is a subset of the extension of *y*.
- The rdfs:subClassOf's extension is transitive and reflexive.
- If the triplet [<x> <rdfs:subClassOf> <y>] exists, then the extension of *x* is a subset of the extension of *y*.
- If [<x> <rdfs:domain> <y>] and [<u> <x> <v>] exist, then *u* belongs in the extension of *y*.
- If [<x> <rdfs:range> <y>] and [<u> <x> <v>] exist, then *v* belongs in the extension of *y*.

Some RDFS axiomatic triples are shown in Table 2-1.

Table 2-1: RDFS axiomatic triples

<rdf:type> <rdfs:domain> <rdfs:Resource>
<rdfs:domain> <rdfs:domain> <rdf:Property>
<rdfs:range> <rdfs:domain> <rdf:Property>
<rdfs:subPropertyOf> <rdfs:domain> <rdf:Property>
<rdfs:subClassOf> <rdfs:domain> <rdfs:Class>
<rdf:type> <rdfs:range> <rdfs:Class>
<rdfs:domain> <rdfs:range> <rdfs:Class>
<rdfs:range> <rdfs:range> <rdfs:Class>
<rdfs:subPropertyOf> <rdfs:range> <rdf:Property>
<rdfs:subClassOf> <rdfs:range> <rdfs:Class>

2.2 The RDF Query Language (RQL)

RQL[5] ([Kar00]) is a typed declarative query language for RDF. It is defined by a set of basic queries and iterators that can be used to build new ones through functional composition; it can combine schema paths for executing complicated schema navigations; not many languages support this type of queries. However, its major innovation lies in its ability to ask queries both on the schema and data levels. It

[5] For the complete RQL syntax, formal semantics and type inference rules, readers are referred to the RQL online documentation found at: http://139.91.183.30:9090/RDF/RQL/

supports *generalised path expressions* featuring variables on labels for both classes and properties, i.e. nodes and arcs in the graph representation, respectively. Finally, it provides set-theoretic operators, allows using XML Schema data types, aggregate functions and arithmetic operations on data values.

```
<?xml version="1.0"?>
<rdf:RDF ...>
    <rdf:Property rdf:ID="flyAtSpeed">
        <rdfs:domain rdf:resource="#Bird"/>
        <rdfs:domain rdf:resource="#Plane"/>
        <rdfs:range rdf:resource="datatype:Integer/>
    </rdf:Property>
</rdf:RDF>
```

Figure 2-3: Definition of a property with multiple domains

2.2.1 Differences between RDF and RQL Semantics

RQL relies on a type system that slightly differs from the axiomatic foundation adopted by the RDF and RDFS specifications. Moreover, RQL provides additional constraints to those offered by RDF Model Theory. More precisely, the RQL type system and semantics *(i)* make a clear distinction between the different *RDF/S* abstraction layers (data, schema and metaschema), *(ii)* enforce that a property's domain and range are always defined and unique (see Figure 2-3), *(iii)* do not allow the existence of cycles in the class and property hierarchies (defined using the rdfs:subClassOf and rdfs:subPropertyOf properties, respectively), *(iv)* state that the set inclusion of the domain and range are preserved for specialised properties (see Figure 2-4), *(v)* do not consider literal types as classes, *(vi)* do not allow the use of typed literals in statements in the data layer and *(vii)* demand that in each statement the subject and object resources should be (direct or indirect) instances of the domain and range classes of the property, respectively.

The first constraint has been introduced to define the appropriate interpretation functions that allow passing from one abstraction layer to another. The result of this distinction is that *(a)* a class must be instance of a metaclass of classes (see Figure 2-5), *(b) subsumption* relations are not allowed between classes and metaclasses and *(c)* a metaclass cannot be instance of some other node, since abstraction layers higher than the metaschema are not defined.

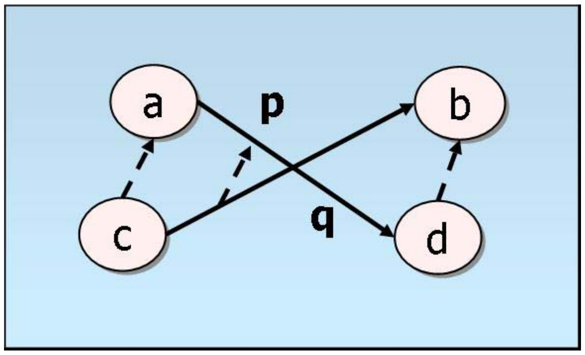


Figure 2-4: Definition of a subproperty not preserving set inclusion of the range

The second constraint has been introduced to clarify the semantics of properties since, when the optional declaration of multiple domains and constraints is permitted, properties may have as value both resources and literals. This might result in semantic inconsistencies, since URIs identify resources, while values identify literals.

The introduction of cycles is, finally, prohibited because they may considerably affect the manipulation of already created RDF/S schemas and resource descriptions. Moreover, the fourth constraint ensures that the domains and ranges of subproperties are subclasses of the ones of their super-properties. If this constraint is not issued, the existence of cycles can indirectly be implied.

Example 2.1: Figure 2-4 states that d is a subclass of b, which means that every instance of class d is an instance of class b. Moreover, it states that p is a subproperty of q, which means that the property extent of p is a subset of q's. This stands only when the resources appearing in the property extent of p that are instances of its range (and domain), namely b, appear in the property extent of q, therefore, are instances of its range (and domain), namely d. Thus, there is a subset b_s of b's extent and a subset d_s of d's extent such that every resource in d_s is, also, in b_s and reversely; a cycle is implied. The subset b_s encompasses all resources used as objects in statements involving the property q. ■

```
<?xml version="1.0"?>
<rdf:RDF ...>
    <rdf:Class rdf:ID="Animal"/>
    ...
    <ms:Animal rdf:ID="Woodpecker"/>
    ...
    <ms:Woodpecker rdf:ID="Woody"/>
    ...
</rdf:RDF>
```

Figure 2-5: Definition of a class as instance of another class

Not considering typed literals at the data layer does not rule out the use of types for literals. Since every property must have a unique range defined, the type restriction of literals can be provided from the range of the corresponding property.

2.2.2 Basic Queries

The basic RQL queries constitute the building blocks on which more complicated RQL queries are built. They essentially provide the means to access and browse through RDF description bases with minimal knowledge of the employed schema(s). RQL provides a number of functions (see Table 2-2) in order to navigate through an RDF/S schema. For example, the *domain* and *range* functions can be used to retrieve a property's definition (its domain and range), while *subclassOf* and *subPropertyOf* can be used to explore the class and property hierarchies, respectively.

Every RDF/S description base can be viewed as a graph, i.e. as a collection of nodes and edges. Thus, the basic queries gaining access to the data layer of such graphs are formed by the appropriate schema names.

Example 2.2: The query

Artist

returns a bag containing all resources of type *Artist*, i.e. those resources belonging to its *class extent*. ■

Table 2-2: Function examples

Basic Query Function	**Result**
domain(creates)	Returns the domain class of property *creates*, i.e. *Artist*
range(creates)	Returns the range class of property *creates*, i.e. *Artifact*
subClassOf^(Artist)	Returns a bag containing the direct subclasses of class *Artist*, i.e. *Painter*, *Sculptor*
subPropertyOf(creates)	Returns a bag containing the subclasses of property *creates*, i.e. *paints*, *sculpts*
Namespace(Artist)	Returns the namespace where class *Artist* is defined, i.e. http://www.icom.com/schema.rdf

In order to get the *proper extent* of a class (or property), meaning only the nodes (edges) of the graph labelled with the class (property) name, the symbol '^' must be used.

Example 2.3: Likewise, the query

creates

returns a bag of ordered pairs of resources belonging to the *extended interpretation* of *creates*, i.e. its *property extent*. ■

Note that the schema nodes and edges (i.e. the RDF/S schema) can, also, be queried as normal data using metaclass names. The core RDF metaclasses *Class* and *Property* can be used to retrieve the names of all classes and properties, respectively. Other basic query functions are *namespace*, which can be used to retrieve a namespace, standard theoretic set operators (*union*, *minus*, *intersect*), which can be applied on collections of the same type and the aggregate functions *min*, *max*, *avg*, *sum* and *count*.

2.2.3 Composite Queries

RQL supports the *SELECT-FROM-WHERE* filters in a similar way as they appear in SQL queries. The filters combine the basic queries presented above and generalised path expressions with variables on nodes and edges to traverse RDF/S description graphs at arbitrary depths.

The result of an RQL filter is an RDF Bag container value on which iterators can be defined using nested queries, while ordered tuples can be represented by RDF Sequences and be accessed through position indexes. As in SQL queries, the SELECT clause states which variables' values are projected in the result and constructs ordered tuples for them. The FROM clause consists of path expressions that define the part of the RDF/S graph that will participate in the evaluation of the query. Each path expression corresponds to a series of steps. Each step represents movement in a particular direction by identifying node labels and can apply one or more predicates to eliminate nodes that fail to satisfy a given condition. These filtering conditions are declared at the optional WHERE clause. The result of each step is a list of nodes that serves as a starting point for the next step. Moreover, the optional clause NAMESPACE can be used to define prefixes.

The generalised path expressions allow navigating throughout *(i)* the schema, *(ii)* the data, or *(iii)* both. Furthermore, the path expressions are used to navigate either based on classes or properties. The basic RQL path expressions are illustrated in Table 2-3 and Table 2-4. Note that all path expressions appear in their general form where variables are not assigned to constant values. The same interpretations are used when variables are valuated with constants; the only difference is that they get extended with the appropriate equalities between variables and constants. More variations of the basic path expressions can be created using the symbol '^' on the paths used for data and mixed navigation. The examples to follow will illustrate the above functionality.

2.2.3.1 Schema Navigation

The schema navigation involves exploring taxonomies of classes and properties using appropriate conditions. Take for example the path expression $\{\$C_1\}@P\{\$C_2\}$ along with the condition $@P = p$, where $\$C_1$ and $\$C_2$ are class variables and *@P* is a property variable. This path allows finding all related schema classes for the given property *p*; *C_1* and *C_2* iterate over *subClassOf(domain(p))* and *subClassOf(range(p))*, respectively. If we want to retrieve all related schema properties for a specific class, the path *{$C}@P* (*@P{$D}*) along with the condition *$C = ...* (*$D = ...*) can be used. For each valuation *p* of *@P*, the class variable *$C* (*$D*) ranges over *subClassOf(domain(p))* (*subClassOf(range(p))*); the results are filtered and only those properties satisfying the condition for *$C* (*$D*) are kept. More complex schema navigation can take place by combining the path expressions.

Table 2-3: Basic RQL class path expressions and their interpretation

Path Expression	Interpretation
$C	{*c* | *c* is a schema class}
$C{$D}	{[*c, d*] | *c*, *d* are schema classes, *d* is a subclass of *c*}
$C{X}	{[*c, x*] | *c* a schema class, *x* in the extended interpretation of class *c*}
$C{X; $D}	{[*c, x, d*] | *c, d* are schema classes, *d* is a subclass of *c*, *x* is in the extended interpretation of *d*}

Example 2.4: The query

```
SELECT    $D, @P2
FROM      creates{$D}.@P2
```

is equivalent to the query

```
SELECT    $D, @P2
FROM      @P1{$D}, {$D}@P2
WHERE     @P1 = creates
```

For each valuation of *$D* based on the first path, the second path gets evaluated. ■

2.2.3.2 Data Navigation

Sometimes we are interested in browsing RDF description bases without taking into account the domain and range restrictions imposed by schema properties. This is the case data navigation is used. There are a number of generalised path expressions that can be used for this kind of navigation. For example, the path *$C{X}* along with the constraint *$C = c* can be used to retrieve the extended interpretation (extent) of a specific class *c*. Likewise, the path *{X}@P{Y}, @P = p* retrieves the extended interpretation of the given property *p*. Like in schema navigation, the path expressions can be combined creating complex queries.

Table 2-4: Basic RQL property path expressions and their interpretations

Path Expression	Interpretation
@P	{*p* | *p* is a schema property}
{$C}@P{$D}	{[*c, p, d*] | *p* is a schema property, *c*, *d* are schema classes, *c* is a subclass of *p*'s domain, *d* is a subclass of *p*'s range}
{X}@P{Y}	{[*x, p, y*] | *p* a schema property, [*x, y*] in the extended interpretation of *p*}
@P{Y; $D}	{[*p ,y, d*] | *p* is a schema property, *d* is a schema class, *d* is a subclass of *p*'s range, *y* is in the extended interpretation of *d*, ∃*x* [*x, y*] is in the extended interpretation of *p*}
{X}@P{$D}	{[*x, p, d*] | *p* is a schema property, *d* is a schema class, *d* is a subclass of *p*'s range, ∃*y* in the extended interpretation of *d*, [*x, y*] is in the extended interpretation of *p*}
{X; $C}@P{Y; $D}	{[*x, c, p, y, d*] | *p* is a schema property, *c*, *d* are schema classes, *c* is a subclass of *p*'s domain, *d* is a subclass of *p*'s range, *x* is in the extended interpretation of *c*, *y* is in the extended interpretation of *d*, [*x, y*] is in the extended interpretation of *p*}

Example 2.5: The RQL query

```
SELECT    X, Y
FROM      Painter{X}.creates{Y}
```

which is equivalent to

```
SELECT      X, Y
FROM        $C{X}, {X}@P{Y}
WHERE       $C = Painter and @P = creates
```

is a complex query example. ■

When the paths used for data navigation get extended with the use of '^', only proper interpretations of classes (properties) are considered.

Example 2.6: In order to catch the proper interpretations the last example gets rewritten as

```
SELECT      X, Y
FROM        ^Painter{X}.^creates{Y} ■
```

2.2.3.3 Mixed Navigation

RQL allows the combination of schema and data filtering and navigation through the use of mixed path expressions. Thus, queries like the one in the following example can be posed.

Example 2.7: The query

```
SELECT      *
FROM        {X; $C}creates{Y}
```

returns the extended interpretation (extent) of property *creates* while, at the same time, iterates through the subclasses of the domain of *creates* so that *X* is in the extended interpretation of one of them. ■

2.3 RDF View Language (RVL)

RVL ([Mag03]) is a view definition mechanism for the Semantic Web. We choose to support it in our system in order to handle RDF/S views defined for personalisation reasons on top of other RDF/S schemas. People may not always be interested on the global virtual RDF/S schema; therefore, define RVL views over it. Since RQL queries can be posed on the RVL views, we should be able to check them for containment and reformulate them into SQL queries as well.

RVL is based on the RDF/S data model and takes advantage of the expressiveness of RQL. RVL exploits the RQL type system and the abstraction levels of an RDF/S graph to specify two operators that are able to support all the necessary functionality. This is its most important advantage.

Figure 2-6 presents the creation of a virtual RDF/S description schema. Typically, an RVL view is defined as a virtual RDF schema consisting of a set of class and property definitions and the hierarchies defined between them. Practically, an RVL view consists of a set of statements defining parts of the view. A definition statement refers to the creation of new virtual (meta)classes/properties, to the reuse of a set of (meta)classes/properties, to member attribution to the virtual (meta)classes/properties, and to the creation or reuse of subsumption relations between classes (virtual or not) using expressions of the view definition language. Being a virtual namespace, an RVL view gets distinguished by a unique URI given by its creator. This URI constitutes the prefix for the unique identifiers of the virtual structures. An RVL view's definition has the form:

```
[VIEW            operator
FROM             RQL_path_expression
WHERE            filtering conditions
USING NAMESPACE            root_schema_namespace]
[…………..]
USING NAMESPACE            root_schema_namespace]
CREATE NAMESPACE           RVL_view_namespace
```

The *FROM-WHERE-USING NAMESPACE* clauses are used exactly as in RQL. From the newly imported clauses, *VIEW*, when used with one of RVL's operands, creates in the virtual schema constructs of the type specified by itself; *CREATE NAMESPACE* defines the URI of the namespace defined for the view, like is done for a schema's namespace, along with a prefix that will be used as shorthand. This prefix can be used when defining another view. The VIEW clause is the one that makes the difference with RQL; while the SELECT clause is used to define which values will be projected as a result, the VIEW clause defines a virtual RDF schema.

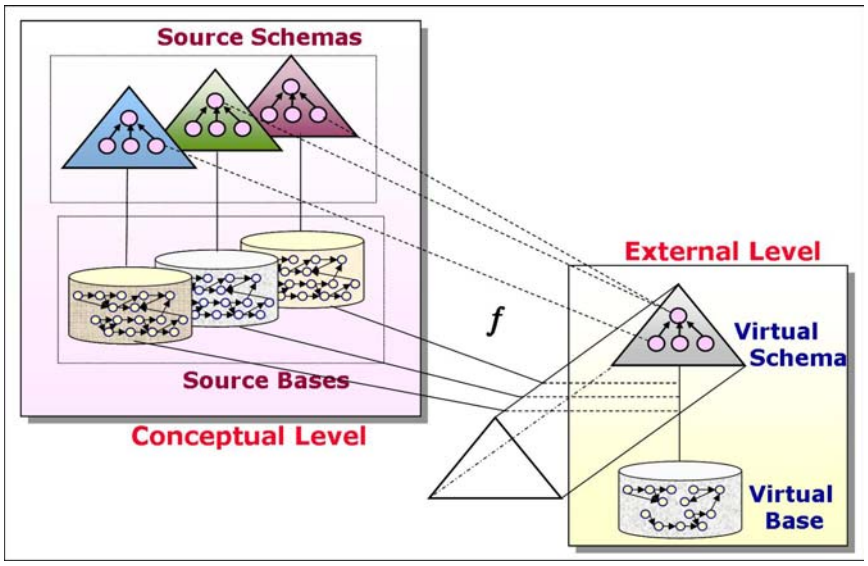


Figure 2-6: RVL view creation process

The two operators supported by RVL are namely the instantiation and the subsumption operators. The instantiation operator is used to state the type of the new construct, i.e. whose instance it is. Its general syntax replaces the definition of the VIEW clause with "*VIEW Symbol(Expression)*".

Example 2.8: The declaration

VIEW rdfs:Class("Artist")
USING NAMESPACE rdfs="&http://www.w3c.org/2000/01/rdf-schema#"

is a very simple example of an RVL view defining a virtual class. ■

The general form of RVL views with the subsumption operator makes use of the VIEW clause: "*VIEW* $Symbol_1$<$Symbol_2$>", which states that $Symbol_2$ is a subclass (subproperty) of $Symbol_1$.

Example 2.9: The declaration

VIEW Person<Artist>

is the simplest example of an RVL view of this type. ■

Moreover, the two operators can be combined to form more complex view definitions.

2.4 Conclusions

Concluding this short introduction to Semantic Web technologies, we can say that RDF/S disposes an expressive, still simple, model that allows describing metadata about web resources, e.g. create hierarchies of classes and properties, and favours reusability of existing descriptions. Moreover, RDF/S is serialised in XML, therefore native to the Web, and a W3C standard, which ensures its wide acceptance.

However, without equivalently simple and expressive query and view definition languages, RDF/S' handiness would be limited. RQL and RVL satisfy this need as they successfully adopt the model of RDF/S. They allow expressing both simple and complex queries and views using syntax similar to SQL, therefore, familiar to the majority of users.

Chapter 3

Semantic Web Logic Framework (SWLF)

The choice of a logic-based framework is crucial in order to support a Semantic Web middleware for optimising and reformulating RQL queries and RVL views. More precisely our goal is to establish a framework that will *(i)* allow reformulating an RQL query to an SQL query using *RDB→RDF mappings*, *(ii)* allow minimising both the RQL queries given as input and the output SQL queries, *(iii)* consider during the reformulations as much information as possible in order to have optimal results – this information comes in the form of (integrity) constraints, either for the RDF/S schemas or the underlying relational database schemas, and in the form of materialised views, either relational or RVL ones – and *(iv)* provide more fundamental features, like checking RQL queries for containment and/or equivalence.

While there has been significant amount of research on *relational query reformulation*, there are not many theoretical foundations on RDF/S query optimisation and reformulation; specifically for RQL queries, there is no background theory at all. Thus, in order to solve our reformulation problem, we come to a dilemma; shall we try to make use of existing work on relational query reformulation or shall we start from scratch? We opt for the first alternative. Therefore, we adopt Linear Datalog, which is a robust formalism, in our Semantic Web Logic Framework (SWLF) in order to be compatible with relational theory and to take advantage of the capability of logical languages to express relationships in generic ways; the latter is indispensable in a semantic integration middleware. So, in SWLF's context, the RDF/S schemas are expressed as Datalog facts and the RDB→RDF mappings, the RQL queries and the relational and RVL views are expressed as Datalog rules.

3.1 Datalog Rules

This way the RQL reformulation/optimisation problem reduces to the relational equivalent. Remember that there is a straightforward correspondence of linear Datalog rules and UNION-SELECT-PROJECT-JOIN (USPJ) relational expressions that are the ones of interest to us (we do not consider nesting, order by, group by and aggregates). This kind of relational expressions can be seen as unions of conjunctive queries.

Definition 3.1: *A query q over a relational schema RS has the form*

$$\bigcup_{i=1}^{l} q(x) :- \varphi_i(x, \psi_i)$$

where x, ψ_i are tuples of variables and φ_i are conjunctions of relational atoms of the form R(ω_1, ..., ω_l) belonging to RS and equality atoms of the form $\omega=\omega'$, where ω_1, ..., ω_l, ω, ω' are variables or constants. ■

As a result, all available background theory on *relational query optimisation* gets exploited; problems like *query containment*, *query composition*, *query rewriting* using views and *query minimisation* have been proven to be algorithmically solvable for the kind of queries that interests us in the presence of certain classes of constraints.

3.2 Constraints

Constraints play a fundamental role in relational theory. They express relationships that must hold between data in relational databases. Thus, they can be used in multiple ways for integrity checking – which is how they got their name "*integrity constraints*", query optimisation via semantics, cooperative answering via semantics, database combination in a semantically consistent manner, etc. ([GGGM98]). They usually come in the form of (primary) keys, which are functional dependencies[6], and foreign keys, which are inclusion dependencies.

For the RQL containment, minimisation, and reformulation problems we both introduce constraints on the relational scenario capturing RDF/S and exploit integrity constraints coming from the underlying relational databases. Thus, we consider a fairly large class of constraints in SWLF in order to fully take advantage of their functionality, namely disjunctive embedded dependencies (DEDs), as were introduced in [Deu02].

[6] [AHV95] provides an analytical classification of first-order (relational) constraints.

Definition 3.2: *A disjunctive embedded dependency has the general form*

$$\forall x\left[\varphi(x) \rightarrow \bigvee_{i=1}^{l} \exists \psi_i \varphi_i^{'}(x,\psi_i)\right]$$

where x, ψ_i are tuples of variables and φ, $\varphi_i^{'}$ are conjunctions of relational atoms of the form $R(\omega_1, ..., \omega_l)$ and equality atoms of the form $\omega=\omega^{'}$, where $\omega_1, ..., \omega_l, \omega, \omega^{'}$ are variables or constants; φ may be the empty conjunction. ■

The constraints are named DEDs after the classical embedded dependencies (EDs) contained when *l=1* ([AHV95]).

The definition of Linear Datalog as the language for representing the RDF/S schemas, the RVL views, the RDB→RDF mappings and the RQL queries is strictly related to the class of constraints considered. Every Datalog fact and rule gets translated into DEDs as will be illustrated later.

3.3 First-order Logic Representation for RDF/S

SWIM's logic-based framework (SWLF) should capture RDF/S semantics and queries, as well as facilitate the expression of the mappings. Thus, a set of *first-order relations*[7] (predicates) is used for representing RDF/S schemas along with a set of *first-order constraints* preserving the RDF/S semantics.

3.3.1 RDF/S Predicates

Definition 3.3: *The first-order schema for describing RDF/S documents is a set R of relations, where R={CLASS, PROP, C_SUB, P_SUB, C_EXT, P_EXT}. Each relation R_i consists of a set A_i of attributes as shown in Table 3-1. In order to enhance clarity three basic types are used forming the set T={resource, property, class}. Each attribute A_{ij} has assigned to itself one type T_i.* ■

The intuition behind the relations is pretty much obvious:

- CLASS(c) iff *c* is an RDF/S schema class
- PROP(c, p, d) iff *p* is a RDF/S schema property with domain *c* and range *d*

[7] The terms 'first-order' and 'relational' are used alternatively. This is possible due to the correspondence between relational expressions and first-order logic queries (FOL) as stated in [CM77].

- C_SUB(d, c) iff *d* is a subclass of *c*
- P_SUB(q, p) iff *q* is a subproperty of *p*
- C_EXT(c, x) iff the resource *x* is in the proper extent (i.e., it is a direct instance) of *c*
- P_EXT(x, p, y) iff the pair *(x, y)* is in the proper extent (i.e., it is a direct instance) of *p*.

It should be noted that class extents do not have to be disjoint, i.e., they might overlap. The same goes for property extents.

Table 3-1: First-order schema for RDF

Relation	**Type**
CLASS	*Set⟨name: Class⟩*
C_SUB	*Set⟨subC: Class, class: Class⟩*
C_EXT	*Set⟨class: Class, inst: Resource⟩*
PROP	*Set⟨subject: Class, predicate: Property, object: Class⟩*
P_SUB	*Set⟨subP: Property, prop: Property⟩*
P_EXT	*Set⟨subject: Resource, predicate: Property, object: Resource⟩*

The relations CLASS, PROP, C_SUB and P_SUB are used for representing an RDF/S schema and, thus, constitute Datalog facts. On the contrary, the C_EXT and P_EXT relations represent the RDF/S resource descriptions and are used in the RDB→RDF mapping rules. These mappings populate the C_EXT and P_EXT relations, i.e. state how the underlying relational data can be published as RDF data.

3.3.2 RDF/S Constraints

Although the above predicates capture successfully RDF/S, they provide no information about the semantics of RDF/S schemas and resource descriptions; they cannot ensure that a valid RDF/S schema is being modelled. Thus, a set of DEDs, namely δ_{Mod}, has been adopted for stating and preserving RDF/S semantics. These constraints can be separated in three categories: *basic constraints*, *sub* (hierarchy) *constraints* and *domain-range constraints*.

Table 3-2: Basic constraints

Description	Formal Definition
➊ Every resource in the extent of a class implies the existence of the corresponding class	$\forall c, x\ C\_EXT(c, x) \rightarrow CLASS(c)$
➋ The subclass relationship relates classes	$\forall c, d\ C\_SUB(d, c) \rightarrow CLASS(c) \wedge CLASS(d)$
➌ The domain & range of every property is a class	$\forall c, p, d\ PROP(a, p, b) \rightarrow CLASS(a) \wedge CLASS(b)$
➍ The domain & range of every property is unique	$\forall a, p, b, c, d\ PROP(a, p, b) \wedge PROP(c, q, d) \wedge p = q \rightarrow a = c \wedge b = d$
➎ Every statement in the extent of a property implies the existence of the corresponding property	$\forall x, p, y\ P\_EXT(x, p, y) \rightarrow \exists c, d\ PROP(c, p, d)$
➏ The subproperty relationship relates properties	$\forall p, q\ P\_SUB(p, q) \rightarrow \exists a, b, c, d\ PROP(a, p, b) \wedge PROP(c, q, d)$

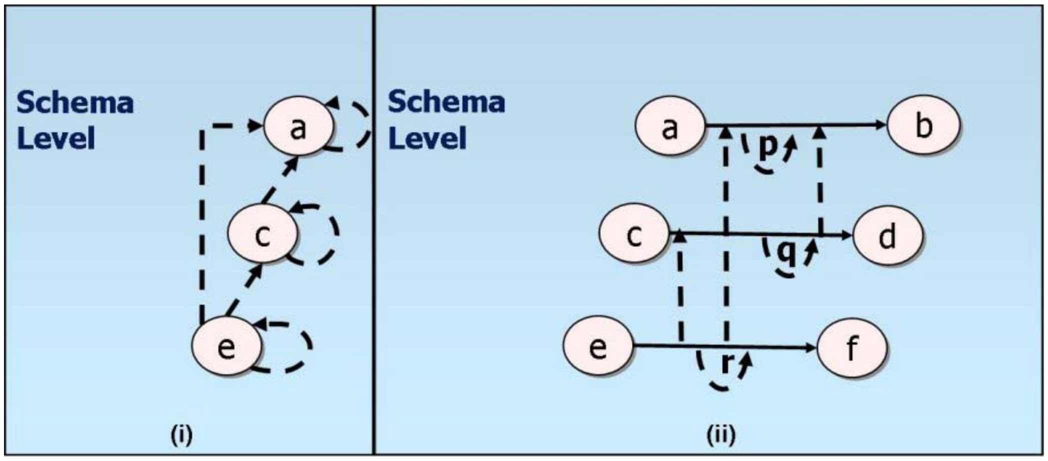


Figure 3-1: Class/property reflexivity/transitivity

3.3.2.1 Basic Constraints

This category hosts some general constraints (see Table 3-2), which are more or less obvious.

Table 3-3: SUB constraints

Description	Formal Definition
❶ Every class is a subclass of itself (reflexivity) (Figure 3-1i)	*∀c CLASS(c) → C_SUB(c, c)*
❷ The subclass relationship is transitive (Figure 3-1i)	*∀a, c, e C_SUB(e, c) ∧ C_SUB(c, a) → C_SUB(e, a)*
❸ Every property is a sub-property of itself (reflexivity) (Figure 3-1ii)	*∀c, p, d PROP(c, p, d) → P_SUB(p, p)*
❹ The sub-property relationship is transitive (Figure 3-1ii)	*∀p, q, r P_SUB(p, q) ∧ P_SUB(q, r) → P_SUB(p, r)*
❺A class is both subclass and super-class of itself only (antisymmetry)	*∀a, c C_SUB(c, a) ∧ C_SUB(a, c) → a=c*
❻ A property is both subproperty and super-property of itself only (antisymmetry)	*∀p, q P_SUB(q, p) ∧ P_SUB(p, q) → p=q*

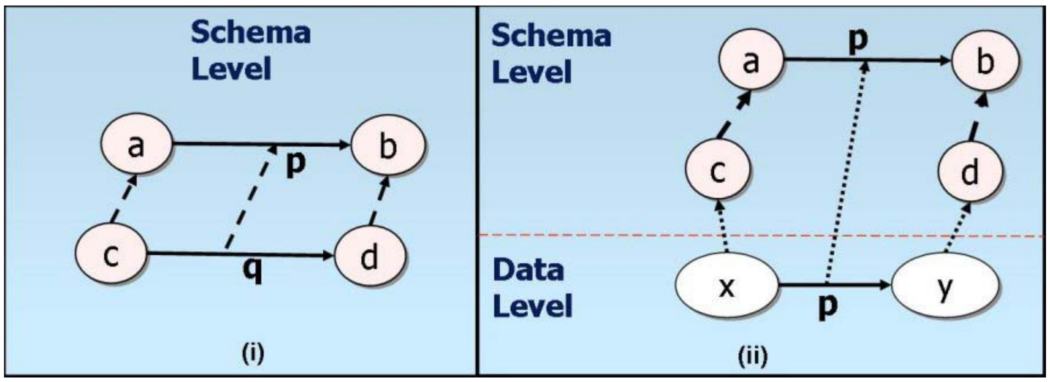


Figure 3-2: Domain-range constraints

3.3.2.2 SUB Constraints

The SUB constraints refer to class/property hierarchy (Table 3-3). Both the C_SUB and P_SUB relations are reflexive, transitive and antisymmetrical.

3.3.2.3 Domain-range Constraints

This category contains two constraints referring to properties' domains and ranges (see Table 3-4).

Table 3-4: Domain-range constraints

Description	Formal Definition
➊ In a valid RDF description schema the domain (range) of every sub-property is subsumed by the domain (range) of its super-property (Figure 3-2i)	$\forall a, p, b, c, q, d\ PROP(a, p, b) \wedge PROP(c, q, d) \wedge P\_SUB(q, p) \rightarrow C\_SUB(c, a) \wedge C\_SUB(d, b)$
➋ In a valid RDF description base the subject/object resources in every statement are (direct or indirect) instances of the property's domain/range classes (Figure 3-2ii)	$\forall a, p, b, x, y\ PROP(a, p, b) \wedge P\_EXT(x, p, y) \rightarrow \exists c, d\ C\_SUB(c, a) \wedge C\_SUB(d, b) \wedge C\_EXT(c, x) \wedge C\_EXT(d, y)$

3.4 Differences between SWLF and RDF/S Semantics

SWLF succeeds in almost fully capturing the RDF/S type system and semantics adopted from RQL (see subsection 2.2.1). Thus, it presents almost the same differences to the RDF/S Model Theory as RQL Semantics does. The only difference rises in the handling of literal datatypes. According to RDF/S Model Theory, literal datatypes are classes. Nevertheless, while in RQL they are not considered as classes, in SWLF they are. However, in contrast to RDF/S Model Theory, datatype classes are not considered to be subclasses of class rdfs:Literal.

3.5 From RDF/S Schemas to SWLF

Having defined SWLF, the first step that needs to be taken for checking either RQL query containment or reformulating/minimising an RQL query into SQL queries is passing from the RDF/S schema to SWLF. This means that all information concerning an RDF/S schema has to be translated in terms of the CLASS, PROP, C_SUB and P_SUB relations. The procedure is rather straightforward:

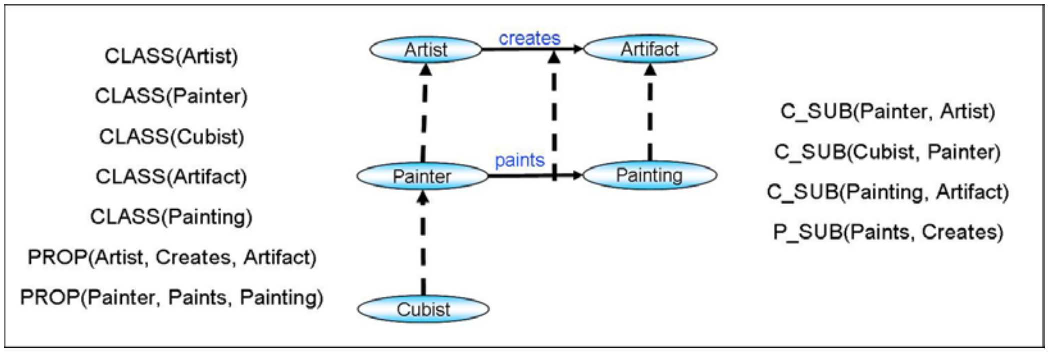


Figure 3-3: An RDF/S schema and its SWLF translation

- For every RDF/S class *c*, the fact *CLASS(c)* is true and is added in the Datalog program
- For every RDF property *p* having as domain class *c* and range class *d*, the fact *PROP(c, p, d)* is true and gets added in the Datalog program
- For every class *d* that is connected to a class *c* through the property *rdfs:subClassOf*, the fact *C_SUB(d, c)* is true and gets added in the Datalog program
- For every property *q* that is connected to a property *p* through the property *rdfs:subPropertyOf*, the fact *P_SUB(q, p)* is true and gets added in the Datalog program

Example 3.1: Following the above procedure the RDF/S schema introduced in Figure 3-3 translates into the SWLF facts presented in the same figure. ■

3.5.1 Translating the Facts into Constraints

The algorithms for solving the containment (see section 4.1) and minimisation (see section 4.3) problems demand that all information (except the query) is passed as input in the form of constraints. Thus, the next step for SWIM is to translate the Datalog facts describing the RDF/S schema into DEDs. In order to fully capture the intended meaning of the facts, for each SWLF predicate modelling RDF/S schemas (CLASS, PROP, C_SUB, P_SUB) one constraint universally quantifying the predicate's variables and many existentially quantifying them are needed[8]. The existentially quantifying constraints advertise the existence of the RDF/S schema's classes, properties and hierarchy relationships, while the universally quantifying ones state that these are the only classes, properties and hierarchy relationships needed to describe the RDF/S schema.

$$\forall name\ (CLASS(name) \rightarrow name=\text{“Artist”} \vee name=\text{“Painter”} \vee name=\text{“Cubist”} \vee name=\text{“Artifact”} \vee name=\text{“Painting”})$$

$$\exists name\ CLASS(name) \wedge name=\text{“Artist”}$$

$$\exists name\ CLASS(name) \wedge name=\text{“Painter”}$$

$$\exists name\ CLASS(name) \wedge name=\text{“Cubist”}$$

$$\exists name\ CLASS(name) \wedge name=\text{“Artifact”}$$

$$\exists name\ CLASS(name) \wedge name=\text{“Painting”}$$

$$\forall subP\ \forall prop\ (P\_SUB(subP, prop) \rightarrow (subP=\text{“creates”} \wedge prop=\text{“creates”}) \vee (subP=\text{“paints”} \wedge prop=\text{“creates”}) \vee (subP=\text{“paints”} \wedge prop=\text{“paints”}))$$

$$\exists\ subP \exists prop\ P\_SUB(subP, prop) \wedge subP=\text{“creates”} \wedge prop=\text{“creates”}$$

$$\exists\ subP \exists prop\ P\_SUB(subP, prop) \wedge subP=\text{“paints”} \wedge prop=\text{“creates”}$$

$$\exists\ subP \exists prop\ P\_SUB(subP, prop) \wedge subP=\text{“paints”} \wedge prop=\text{“paints”}$$

Figure 3-4: Constraints for the CLASS, P_SUB relations of Figure 3-3

With respect to the general form of DEDs, each universally quantifying constraint has the form

$$\forall x \left[\varphi(x) \rightarrow \bigvee_{i=1}^{l} \varphi_i^{'}(x) \right]$$

where φ is the SWLF predicate and each $\varphi_i^{'}$ is a conjunction of equality atoms. Each existentially quantifying constraint has the form

$$\exists\, \psi \varphi^{'}(\psi)$$

where $\varphi^{'}$ is a conjunction of one relational atom and possibly many equality atoms. The latter form is equivalent to

$$TRUE \leftarrow \varphi^{'}(\psi)$$

where ψ is a tuple of constants. Throughout this thesis we use the former notation.

[8] In the rest of this thesis these constraints will be referred to as universally (existentially) quantifying constraints.

The constraints for C_SUB (P_SUB) take under consideration, additionally to the facts, all the C_SUB (P_SUB) tuples following from the relation's reflexivity and transitivity. Figure 3-4 illustrates the constraints corresponding to part of the RDF/S schema of Figure 3-3 (only the CLASS, P_SUB predicates are considered).

3.6 From RQL Queries to SWLF

Conjunctive RQL queries – i.e. *SELECT-FROM-WHERE* ones – not using aggregate functions, nesting and negation can be seen in a rule-based formalism, which is compatible to Datalog; the only difference is that instead of first-order predicates, RQL path expressions[9] are used. In the rest of this thesis we will focus on this fragment of RQL, namely RQL_{CONJ}, extended with union (RQL_{UCQ}).

Definition 3.4: *An RQL_{CONJ} query has the general form: $ans(\bar{U})$:- ..., $E_i(\bar{U}_i)$, ..., $U_{im}=U_{jn}$, ... The rule's head consists of the query's name ans and the tuple $\bar{U}$ of the returned variables; the rule's body consists of a conjunction of RQL patterns $E_i(\bar{U}_i)$ and equalities $U_{im}=U_{jn}$ between variables and/or constants. Each $\bar{U}_i$ involves the variables X_i, $\$C_i$, $@P_i$, Y_i, $\$D_i$ – where $@P_i$ is a property variable, $\$C_i$ and $\$D_i$ are class variables, X_i and Y_i are resource variables, as we have already seen – or a subset of them.* ■

The above definition of RQL_{CONJ} queries extends in order to comprise queries involving union.

Definition 3.5: *An RQL_{UCQ} has the general form*

$$ans(\bar{U}) :- \bigcup_k body_k$$

where $ans(\bar{U})$ and $body_k$ are given from the definition above. ■

The passing from RQL_{UCQ} queries in the SELECT-FROM-WHERE formalism into RQL_{UCQ} queries in the rule formalism, which facilitates their translation to SWLF, demands that a normalisation phase edges in reducing the

[9] In the rest of this thesis the terms 'RQL path expression' and 'RQL pattern' are used alternately since they are synonymous.

complex path expressions found in the FROM clause into the general form of RQL_{UCQ} queries.

Example 3.2: Observe the RQL_{UCQ} query

SELECT	*X*
FROM	*{X}paints.exhibited{Z}*
WHERE	*Z="http://www.louvre.fr"*

Initially, the normalisation reduces it to the equivalent query

SELECT	*X*
FROM	*{X}paints{Y}, {Y}exhibited{Z}*
WHERE	*Z="http://www.louvre.fr"*

By replacing the constants found in the patterns with variables and adding the corresponding equalities we get:

SELECT	*X*
FROM	*{X}@P_1{Y}, {Y}@P_2{Z}*
WHERE	*@P_1=paints and @P_2=exhibited and Z= "http://www.louvre.fr"*

It is easy to derive the rule that is equivalent to the above RQL query:

ans(X) :- {X}@P_1{Y}, {Y}@P_2{Z}, @P_1=paints, @P_2=exhibited, Z="http://www.louvre.fr" ■

3.6.1 From RQL Patterns to SWLF

When the RQL_{UCQ} query has been translated in a rule-based formalism, the only step remaining for passing to a Datalog rule is replacing the patterns with first-order predicates; the goal is to express the parts of the RDF/S graph that participate in the evaluation of the query, as defined using the patterns, in terms of the SWLF relations. Thus, a translation for each RQL pattern to SWLF is needed based on RQL patterns' semantics and SWLF semantics.

Definition 3.6: *RQL class patterns, i.e. those facilitating navigation through a schema's classes, are translated in SWLF as shown in Table 3-5.* ■

Example 3.3: The class pattern *\$C(X)* translates to *C_SUB(d, c), C_EXT(d, x)*. Remember that *C_EXT(d, x)* returns only the direct instances of class *c*; this is why the *C_SUB(d, c)* relation is introduced: to iterate through the subclasses of class *c*. ■

Definition 3.7: *RQL property patterns, i.e. those facilitating navigation through a schema's properties, are translated in SWLF as shown in Table 3-6*[10]. ■

Example 3.4: Using the translations the example presented above (section 3.6) translates to:

ans(x) :- P_SUB(q_1, p_1), P_EXT(x, q_1, y), P_SUB(q_2, p_2), P_EXT(y, q_2, z), p_1="paints", p_2="exhibited", z="www.louvre.fr" ■

Table 3-5: Class patterns' translation into SWLF

Class Pattern	SWLF Translation
\$C *^\$C*	*CLASS(c)*

\$C{\$D}
^\$C{\$D}
	C_SUB(d, c)

\$C{X}	*C_SUB(d, c), C_EXT(d, x)*
^\$C{X}	*C_EXT(d, x)*
\$C{X; \$D}	*C_SUB(d, c), C_SUB(e, d), C_EXT(e, x)*
^\$C{X; \$D}	*C_SUB(d, c), C_SUB(e, d), C_EXT(e, x), C_EXT(c, x)*
\$C{X; ^\$D}	*C_SUB(d, c), C_EXT(d, x)*
^\$C{X; ^\$D}	*C_SUB(d, c), C_EXT(c, x), C_EXT(d, x)*

Concluding this chapter and having in mind the relational schema *R* and the constraints capturing RDF/S semantics (δ_{Mod}) we can redefine the terms *description base* and *description schema* introduced in [ACK+02]. The new definitions will be considered throughout the rest of the thesis.

Definition 3.8: *An RDF/S description schema DS in SWLF is an instantiation of the relational schema R_S={CLASS, PROP, C_SUB, P_SUB} satisfying δ_{Mod}.* ■

[10] Not all property patterns' translations appear in the table. See Appendix I for the complete list.

Definition 3.9: *An RDF/S description base DB in SWLF given a DS is an instantiation of the relational schema R_B={C_EXT, P_EXT} satisfying δ_{Mod}.* ■

Table 3-6: Property patterns' translation into SWLF

Property Pattern	SWLF Translation
@P *^@P*	*PROP(a, p, b)*

{\$C}@P{\$D}
{\$C}^@P{\$D}
	PROP(a, p, b), C_SUB(c, a), C_SUB(d, b)

{X}@P{Y}
{X}@P
@P{Y}
	P_SUB(q, p), P_EXT(x, q, y)

{X}^@P{Y}
{X}^@P
^@P{Y}
	P_EXT(x, p, y)

{X}@P{Y; \$D}
{X}@P{\$D}
@P{Y; \$D}
	PROP(a, p, b), P_SUB(q, p), P_EXT(x, q, y), C_SUB(d, b), C_SUB(f, d), C_EXT(f, y)

{X}^@P{Y; ^\$D}
^@P{Y; ^\$D}
	PROP(a, p, b), P_EXT(x, p, y), C_SUB(d, b), C_EXT(d, y)

{X ; \$C} @P{Y ; \$D}
{X ; \$C} @P{\$D}
{\$C} @P{Y ; \$D}
	PROP(a, p, b), P_SUB(q, p), P_EXT(x, q, y), C_SUB(c, a), C_SUB(d, b), C_SUB(e, c), C_SUB(f, d), C_EXT(e, x), C_EXT(f, y)

{X ; ^\$C}^ @P{Y ; ^\$D}	*PROP(a, p, b), P_EXT(x, p, y), C_SUB(c, a), C_SUB(d, b), C_EXT(c, x), C_EXT(d, y)*

In the rest of this thesis we will refer to two sets of constraints, namely δ_{RDF} and Δ_{RDF}, depending on the problem we want to solve; both are based on the constraints preserving RDF/S semantics (δ_{Mod}) and those extracted from an RDF/S description schema.

Definition 3.10: *Δ_{RDF} is the set of DEDs formed from δ_{Mod} and all the constraints extracted from a specific RDF/S description.* ■

Definition 3.11: *δ_{RDF} is a set of disjunction-free DEDs. It is the subset of Δ_{RDF} that does not include the universally quantifying constraints.* ■

Chapter 4

RQL Query Optimisation

Throughout this chapter we will study the problems of RQL query containment, equivalence and minimisation using two algorithms, the Chase and the Backchase. We have managed to identify several subcases of these problems by adjusting the fragment of RQL considered to the input information given in the form of constraints. We consider three sets of constraints (δ_{Mod}, δ_{RDF}, Δ_{RDF}) and two fragments of RQL ($\mathrm{RQL}_{\mathrm{UCQ}}$ and its subset $\mathrm{RQL}_{\mathrm{CORE}}$).

In section 4.1 we study the containment problem for $\mathrm{RQL}_{\mathrm{UCQ}}$ queries (subsection 4.1.2), where we have full knowledge of the RDF/S schema - provided by Δ_{RDF}, and proceed with the same problem for $\mathrm{RQL}_{\mathrm{CORE}}$ queries (subsection 4.1.3), where we have partial knowledge of the RDF/S schema - provided by δ_{RDF}. Then, we take a quick look over the equivalence problem for the same RQL fragments (section 4.2). We, also, present in section 4.3 how queries originating from both fragments of RQL can get minimised by considering Δ_{RDF} (subsection 4.3.2) and δ_{RDF} (subsection 4.3.3), respectively. Finally, we show how we can simplify (minimise) $\mathrm{RQL}_{\mathrm{UCQ}}$ queries without having knowledge of a specific RDF/S schema (subsection 4.3.4). This is the case where only the knowledge of RDF/S semantics is provided through δ_{Mod}.

4.1 RQL_{UCQ} Query Containment

All problems aforementioned are based on containment of $\mathrm{RQL}_{\mathrm{UCQ}}$ queries. Thus, it has to be dealt first.

Definition 4.1: *A RQL_{UCQ} query Q_1 is contained in a RQL_{UCQ} query Q_2 ($Q_1 \subseteq Q_2$) given an RDF description schema DS iff for every description base DB conforming to DS the result of Q_1 is contained in that of Q_2 ($\forall DB\ Q_1(DB) \subseteq Q_2(DB)$).* ■

In some cases the containment is obvious. This is the case of simple queries that do not involve complex paths.

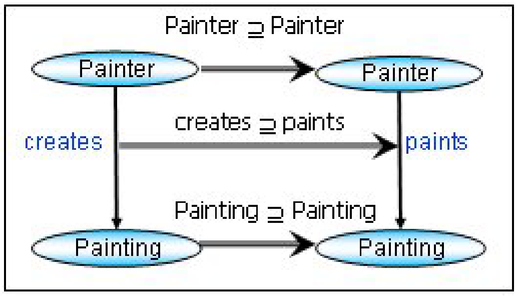


Figure 4-1: Simple graphical containment example

Example 4.1: It is easy to figure out that the query

```
SELECT    X
FROM      {X; Painter}paints{Y; Painting};
```

is contained in

```
SELECT    X
FROM      {X; Painter}creates{Y; Painting};
```

because it is pretty straightforward from the RDF/S semantics that when "a painter paints a painting", at the same time he creates a painting.

Alternatively, in this example, the containment can easily be spotted if the two RQL_{UCQ} queries are seen as graphs, where the nodes correspond to classes and the edges to properties. The subject node of the second query subsumes (is a superclass of) the corresponding one of the first query. The same goes for the object nodes, too.

Moreover, the property edge of the second query subsumes (is a superproperty of) the corresponding one of the first query. Thus, all instances satisfying the first query satisfy the second one, too. Figure 4-1 illustrates the containment check between the two graphs. ■

However, deciding containment of more complex queries is not trivial. There will be, also, cases where even seeing the queries as graphs will not allow checking containment. Moreover, there exist many applications that need dealing with containment in an automated way. Thus, the definition of a sound and complete algorithm for checking RQL_{UCQ} query containment is mandatory. This chapter's goal is to provide such an algorithm that will solve these problems.

4.1.1 Chase Algorithm

The core algorithm of this work is the chase [Deu02]. Chasing a query equals applying a sequence of chase steps to the query. Its handiness originates from the fact that all input information is given in the form of DEDs. The definition of a chase step demands that the notion of homomorphism is familiar to the reader.

Definition 4.2: *A homomorphism from φ_1 into φ_2 is a mapping h from the variables of φ_1 into those of φ_2 such that:*

i. For every equality atom $\omega = \omega'$ in φ_1, $h(\omega) = h(\omega')$ follows from the equality atoms of φ_2 and

ii. For every relational atom $R(\omega_1, \ldots, \omega_l)$ in φ_1, there is an atom $R(\upsilon_1, \ldots, \upsilon_l)$ in φ_2 such that $\upsilon_i = h(\omega_i)$ follows from the equality atoms of φ_2. ■

Definition 4.3: *Given two conjunctive queries $Q_1(x_1, \ldots, x_n) \leftarrow \varphi_1(x_1, \ldots, x_n, y_1, \ldots, y_m)$ and $Q_2(u_1, \ldots, u_n) \leftarrow \varphi_2(u_1, \ldots, u_n, \upsilon_1, \ldots, \upsilon_k)$, where φ_1, φ_2 are conjunctions of relational and equality atoms, a containment mapping from Q_1 to Q_2 is a homomorphism m from φ_1 to φ_2 such that $m(x_i) = u_i$ for $1 \leq i \leq n$.* ■

Definition 4.4: *Let d be a DED (see Definition 3.2), Q be a conjunctive query q(x) :- $\varphi_q(x, \psi)$ (see Definition 3.1) and h be a homomorphism from φ into φ_q. We say that the chase step of Q with d using h is applicable if h allows no extension that is a homomorphism from $\varphi \wedge \varphi_i'$ into φ_q for any $l \geq i \geq 1$. In this case, the result of applying this chase step is the union of queries $\bigcup_i Q_i$, where each Q_i is defined as $q_i(x)$:- $\varphi_q \wedge \varphi_i'(h(x_1), \ldots, h(x_n), f_{i,1}, \ldots, f_{i,ki})$ where the $f_{i,j}$'s are the fresh variables.* ■

The above definition of a chase step ensures that no chase step gets applied when its result is already present in the query.

Example 4.2: The query *Q(x) :- A(x, y), B(x, z)* chases with the DED

$$\forall x \forall y \, A(x, y) \rightarrow V(x)$$

rendering query *Q(x) :- A(x, y), B(x, z), V(x)*. ■

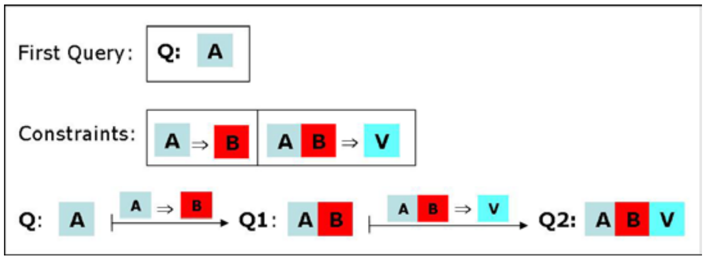


Figure 4-2: Chase illustrative example

When the query has been chased with all available dependencies (constraints) and no more chase steps apply, we say that the chase has ended. The resulting query is called the *universal plan*.

Example 4.3: Figure 4-2 illustrates the basic idea behind chase without taking into account variables. It contains a query and two constraints. The first constraint states that every predicate *A* implies the existence of a predicate *B*. Thus, the query *Q* involving a predicate *A* gets enriched with a predicate *B* resulting in query Q_1. Additionally, the concurrent presence of both a predicate *A* and a predicate *B* implies the existence of a predicate *V* according to the second constraint. Thus, the query Q_1 turns into Q_2 by adding a predicate *V*. Lack of other constraints the chase ends rendering Q_2 as the universal plan of *Q*. ■

Unfortunately, the chase of any conjunctive query with any set of embedded dependencies is not guaranteed to terminate ([AHV95]). Obviously, this result extends for unions of conjunctive queries chasing with DEDs. Thus, appropriate restrictions that will guarantee termination are needed.

4.1.1.1 Stratified-witness

In order to overcome the lack of guarantees for the termination of chase with arbitrary embedded dependencies, [Deu02] identified a property, namely *stratified-witness*; the chase with constraints satisfying this property is guaranteed to terminate. The stratified-witness property is founded on the notion of the *chase flow graph* of a set *C* of constraints.

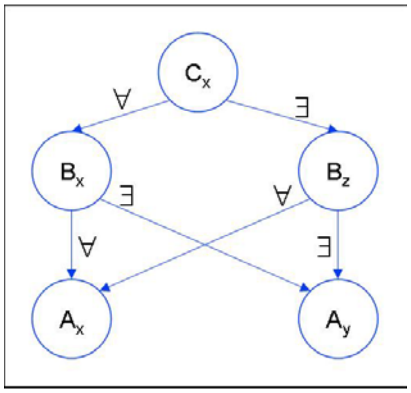


Figure 4-3: Chase flow graph

Definition 4.5: *The chase flow graph G = (V, E) of a set C of constraints is a directed edge-labelled graph whose labels have either the value ∀ or ∃. G is constructed as follows: for every pair of relations R, R' of arities a, a' and every constraint*

$$\forall \vec{x}\left[\ldots\wedge R(u_1,\ldots,u_a)\wedge\ldots\rightarrow\ldots\wedge R'(\upsilon_1,\ldots,\upsilon_{\alpha'})\wedge\ldots\right]$$

in C, E contains the edges $\left(R_i,R_j'\right)_{1\le i\le a,1\le j\le a'}$. *Also, whenever the equality x=y appears in the conclusion of the implication and x, y appear as the* i^{th}, j^{th} *component of R, R', respectively, E contains the edge* $\left(R_i,R_j'\right)$. *Moreover, if for some j, the variable* υ_j *is existentially quantified, the edges* $\left(R_i,R_j'\right)_{1\le i\le a}$ *are labelled with ∃; otherwise they are labelled with ∀.* ■

According to [Deu02] the set *C* of constraints has stratified-witness if none of the cycles in its chase flow graph contains a ∃-labelled edge.

Example 4.4: Given the following two constraints, Figure 4-3 presents their chase flow graph.

$$\forall x\,\forall z\,[B(x, z) \rightarrow \exists y\, A(x, y)]$$
$$\forall x\,[C(x) \rightarrow \exists z\, B(x, z)]$$

It follows from the flow graph that the two constraints satisfy the stratified-witness property. ■

The intuition behind the endless execution of chase steps lack of stratified-witness is simple; each added predicate *A* in the query through a sequence of chase steps (those applied using the constraints that form the cycle) causes the addition of a new predicate *A*, which in its turn causes the addition of another predicate *A*, etc. Although the definition of a chase step prohibits executing steps that will add predicates already present, the introduction of new variables during the chase steps renders the prevention inactive.

Example 4.5: The chase flow graph corresponding to the following constraints contains a cycle with a ∃-labelled edge.

$$\forall x \forall y\, [A(x, y) \rightarrow \exists z\, B(y, z)]$$
$$\forall x\ \forall y\, [B(x, y) \rightarrow A(x, y)]$$

The chase of query $Q(X) :\text{-}\ A(a_1, a_2)$ with them does not terminate as shown below

$Q(X) :\text{-}\ A(a_1, a_2)$
$\rightarrow Q_1(X) :\text{-}\ A(a_1, a_2), B(a_2, a_3)$
$\rightarrow Q_2(X) :\text{-}\ A(a_1, a_2), B(a_2, a_3), A(a_2, a_3)$
$\rightarrow Q_3(X) :\text{-}\ A(a_1, a_2), B(a_2, a_3), A(a_2, a_3), B(a_3, a_4)$
$\rightarrow Q_4(X) :\text{-}\ A(a_1, a_2), B(a_2, a_3), A(a_2, a_3), B(a_3, a_4), A(a_3, a_4)$
$\rightarrow \ldots$ ■

Although embedded dependencies (EDs) were the initial application domain for the stratified-witness property, the latter can be used with disjunction-free DEDs, i.e. DEDs consisting of a single conjunctive query, too. Disjunction-free DEDs differ from EDs in that they allow *(a)* equalities in the left-hand side of the constraints, *(b)* equalities in the right-hand of the constraints involving existentially quantified variables, and *(c)* both variables and constants in the atoms used. We illustrate in Appendix B why this is possible.

Proposition 4.1: *The chase of a conjunctive query Q with a set C of disjunction-free DEDs terminates if the chase flow graph of C has no cycle containing an ∃-labelled edge.* ■

Using the above proposition we get the following definition.

Definition 4.6: *A set C of disjunction-free DEDs satisfying the conditions of the above proposition satisfies the stratified-witness property.* ■

4.1.1.2 Stratified-witness of DEDs

We will start by studying the case of chasing a conjunctive query Q with a finite set C of m DEDs. Each DED d_i can be seen as a union of several disjunction-free EDs d_{ij} whose number n_i is finite, too. The "worst case scenario" is that by combining m d_{ij}'s, one for each i, the chase flow graph of these m d_{ij}'s will contain at least one cycle with at least one ∃-labelled graph. This scenario will take place when the chase steps corresponding to the d_i's whose d_{ij}'s participate in the sequence presented in such a flow graph, get applied. The result will be an infinite number of chase steps. Under any other circumstance, the finite number of DEDs and disjunction-free DEDs in each DED guarantees termination of the chase.

Note that each combination cannot have more than one d_{ij} from each d_i, because there is no way that in a chase sequence the same predicates will trigger twice the same constraint due to the definition of chase step. Since one (or more) predicate(s) produced through chasing some initial predicate(s) cannot trigger the same constraints – this is the case of cycle in the chase flow graph – the chase will terminate.

Proposition 4.2: *The chase of a conjunctive query Q with a set C of DEDs terminates if for all combinations of m d_{ij}'s, one for each i, stratified-witness is preserved. Thus, $O\left(n_{\max}^{m}\right)$ checks for stratified-witness are necessary. (C = {d_i | $1 \leq i \leq m$}, $d_i = \bigvee_{j=1}^{n_i} d_{ij}$, $n_{max}=max(n_i)$)* ■

The following definition follows from the above proposition.

Definition 4.7: *A set C of DEDs satisfying the conditions of the above proposition satisfies the stratified-witness property.* ■

Since a union query consists of a finite number of conjunctive queries, if the DEDs satisfy the stratified-witness property, the chase terminates.

Proposition 4.3: *The chase of a union of conjunctive queries with a set of DEDs satisfying the stratified-witness property terminates.* ■

Example 4.6: Let's see an example where the extension of the stratified-witness for DEDs can be used to locate a set of constraints that may not allow a query's chase to terminate. Imagine the following set *D* of constraints

$(d_1)\ \forall x \forall y\ [A(x, y) \rightarrow B(x, y) \vee C(x, y)]$

$(d_2)\ \forall x \forall y\ [B(x, y) \rightarrow F(x, y) \vee \exists z\ G(x, y, z)]$

$(d_3)\ \forall x \forall y\ [C(x, y) \rightarrow D(x, y) \vee \exists z\ E(y, z)]$

$(d_4)\ \forall x \forall y\ [E(x, y) \rightarrow A(x, y)]$

We have to check eight different chase flow graphs for cycles containing at least one ∃-labelled edge built using the following sub-constraints:

$(d_{11})\ \forall x \forall y\ [A(x, y) \rightarrow B(x, y)]$ $(d_{12})\ \forall x \forall y\ [A(x, y) \rightarrow C(x, y)]$

$(d_{21})\ \forall x \forall y\ [B(x, y) \rightarrow F(x, y)]$ $(d_{22})\ \forall x \forall y\ [B(x, y) \rightarrow \exists z\ G(x, y, z)]$

$(d_{31})\ \forall x \forall y\ [C(x, y) \rightarrow D(x, y)$ $(d_{32})\ \forall x \forall y\ [C(x, y) \rightarrow \exists z\ E(y, z)]$

$(d_4)\ \forall x \forall y\ [E(x, y) \rightarrow A(x, y)]$

The constraints d_{12}, d_{21} (or d_{22}), d_{32} and d_4 create a cycle containing an ∃-labelled edge. Therefore, we cannot guarantee termination of the chase. Let's see now what will happen if the query

ans(x) :- A(x, y)

gets chased with *D*

ans(x) :- A(x, y)

→ ans(x) :- A(x, y), B(x, y)

∨ ans(x) :- A(x, y), C(x, y)

→ ans(x) :- A(x, y), B(x, y), F(x, y) *(no more chase step for this query)*

∨ ans(x) :- A(x, y) , B(x, y), G(x, y, z) *(no more chase step for this query)*

∨ ans(x) :- A(x, y), C(x, y), D(x, y) *(no more chase step for this query)*

∨ ans(x) :- A(x, y), C(x, y), E(y, z)

→ ans(x) :- A(x, y), B(x, y), F(x, y) *(no more chase step for this query)*

∨ ans(x) :- A(x, y) , B(x, y), G(x, y, z) *(no more chase step for this query)*

$\vee$ *ans(x) :- A(x, y), C(x, y), D(x, y)* *(no more chase step for this query)*

$\vee$ *ans(x) :- A(x, y), C(x, y), E(y, z), A(y, z), …*

The chase of the last conjunctive query does not terminate and can go on for ever. ■

4.1.1.3 Chase Steps

Applying the chase in the RDF/S scenario, the chase steps will very often produce unsatisfiable queries, i.e. queries implying equality of distinct variables.

Example 4.7: Chasing query

ans(x) :- P_SUB(q, p), P_EXT(x, q, y), p="paints"

with the universally quantifying constraint for P_SUB in Figure 3-4 using the homomorphism *{subP→q, prop→p}* results in

ans(x) :- P_SUB(q, p), P_EXT(x, q, y), p="paints", q="paints", p="creates"

$\cup$ *ans(x) :- P_SUB(q, p), P_EXT(x, q, y), p="paints", q="paints", p="paints"*

$\cup$ *ans(x) :- P_SUB(q, p), P_EXT(x, q, y), p="paints", q="creates", p="creates"*

The chased query is a union of three conjunctive queries. However, the 1st and the 3rd ones are not valid because *p* is set to have both the values *"paints"* and *"creates"*. Therefore, in practice, the query is equivalent to:

ans(x) :- P_SUB(q, p), P_EXT(x, q, y), p="paints", q="paints" ■

In its original form the chase algorithm does not search after each chase step for unsatisfiable conjunctive queries where equalities between distinct constants are implied. It searches and removes them from the universal plan. However, the algorithm can safely get extended so that such conjunctive queries get removed as soon as they appear, without affecting its soundness and completeness. This algorithm's extension is very useful in the RDF scenario where many such inconsistencies appear while chasing. Throughout this thesis we will refer to the algorithm's extension. Therefore, only the valid queries produced at each step will be presented.

$$\forall source\ \forall name\ \forall dest\ (PROP(source, name, dest) \rightarrow (source=\text{“Artist”} \wedge name=\text{“creates”} \wedge dest=\text{“Artifact”}) \vee (source=\text{“Painter”} \wedge name=\text{“paints”} \wedge dest=\text{“Painting”}))$$

$$\exists source\ \exists name\ \exists dest\ PROP(source, name, dest) \wedge source=\text{“Artist”} \wedge name=\text{“creates”} \wedge dest=\text{“Artifact”}$$

$$\exists source\ \exists name\ \exists dest\ PROP(source, name, dest) \wedge source=\text{“Painter”} \wedge name=\text{“paints”} \wedge dest=\text{“Painting”}$$

$$\forall subC\ \forall clas\ (C\_SUB(subC, clas) \rightarrow (subC=\text{“Artist”} \wedge clas=\text{“Artist”}) \vee (subC=\text{“Painter”} \wedge clas=\text{“Artist”}) \vee (subC=\text{“Painter”} \wedge clas=\text{“Painter”}) \vee (subC=\text{“Artifact”} \wedge clas=\text{“Artifact”}) \vee (subC=\text{“Painting”} \wedge clas=\text{“Artifact”}) \vee (subC=\text{“Painting”} \wedge clas=\text{“Painting”}))$$

$$\exists subC\ \exists clas\ C\_SUB(subC, clas) \wedge subC=\text{“Artist”} \wedge clas=\text{“Artist”}$$

$$\exists subC\ \exists clas\ C\_SUB(subC, clas) \wedge subC=\text{“Painter”} \wedge clas=\text{“Artist”}$$

$$\exists subC\ \exists clas\ C\_SUB(subC, clas) \wedge subC=\text{“Painter”} \wedge clas=\text{“Painter”}$$

$$\exists subC\ \exists clas\ C\_SUB(subC, clas) \wedge subC=\text{“Artifact”} \wedge clas=\text{“Artifact”}$$

$$\exists subC\ \exists clas\ C\_SUB(subC, clas) \wedge subC=\text{“Painting”} \wedge clas=\text{“Artifact”}$$

$$\exists subC\ \exists clas\ C\_SUB(subC, clas) \wedge subC=\text{“Painting”} \wedge clas=\text{“Painting”}$$

Figure 4-4: PROP & C_SUB constraints for the RDF/S schema of Figure 3-3 where class "Cubist" gets ignored

Example 4.8: Given Δ_{RDF} for the RDF/S schema of Figure 3-3, as seen in Figure 3-4 and Figure 4-4, we want to chase the query below. We ignore the class *“Cubist”* for simplicity reasons.

SELECT *X*
FROM *{X}paints{Y}*

translates in SWLF into

ans(x) :- P_SUB(q, p), P_EXT(x, q, y), p=“paints”

which is the query of the previous example. It chases with the 6th basic RDF/S constraint using the elementary homomorphism, i.e. the one that maps each variable to itself, and the result is[11]:

$ans(x) :\!-\ \mathbf{P\_SUB(q, p)}, P\_EXT(x, q, y), \underline{PROP(a, p, b)}, \underline{PROP(c, q, d)}, p=\text{“paints”}$

[11] Bold letters are used to highlight the predicates that trigger a chase step. The predicates added by the same chase step are underlined.

The 3rd basic constraint is applied twice using the homomorphism *{a→c, p→q, b→d}* and the elementary one resulting in:

*ans(x) :- P_SUB(q, p), P_EXT(x, q, y), **PROP(a, p, b)**, **PROP(c, q, d)**, <u>CLASS(a)</u>, <u>CLASS(b)</u>, <u>CLASS(c)</u>, <u>CLASS(d)</u>, p="paints"*

Using the universally quantifying constraint for P_SUB (Figure 3-4) and the homomorphism *{subP→q, prop→p}* we get:

*ans(x) :- **P_SUB(q, p)**, P_EXT(x, q, y), PROP(a, p, b), PROP(c, q, d), CLASS(a), CLASS(b), CLASS(c), CLASS(d), **p="paints"**, <u>q="paints"</u>*

Using the 2nd existentially quantifying constraint for PROP (Figure 4-4) we get:

ans(x) :- P_SUB(q, p), P_EXT(x, q, y), PROP(a, p, b), PROP(c, q, d), <u>PROP(s, r, t)</u>, CLASS(a), CLASS(b), CLASS(c), CLASS(d), p=q="paints", <u>r="paints"</u>, <u>s="Painter"</u>, <u>t="Painting"</u>

By applying the 4th basic constraint (domain-range uniqueness constraint) twice using the homomorphisms *{a→a, p→p, b→b, c→s, q→r, d→t}* and *{a→s, p→r, b→t, c→c, q→q, d→d}*, the query chases to:

*ans(x) :- P_SUB(q, p), P_EXT(x, q, y), **PROP(a, p, b)**, **PROP(c, q, d)**, **PROP(s, r, t)**, CLASS(a), CLASS(b), CLASS(c), CLASS(d), **p=q=r="paints"**, <u>a=c="Painter"</u>, **s="Painter"**, <u>b=d="Painting"</u>, **t="Painting"***

Chasing with the 2nd domain-range constraint (property-class extent compatibility) using the homomorphism {a→c, p→q, b→d} results in:

*ans(x) :- P_SUB(q, p), **P_EXT(x, q, y)**, PROP(a, p, b), **PROP(c, q, d)**, PROP(s, r, t), CLASS(a), CLASS(b), CLASS(c), CLASS(d), <u>C_SUB(e, c)</u>, <u>C_SUB(f, d)</u>, <u>C_EXT(e, x)</u>, <u>C_EXT(f, y)</u>, p=q=r="paints", a=c=s="Painter", b=d=t="Painting"*

Using the 1st domain-range constraint and the elementary homomorphism the query becomes:

*ans(x) :- **P_SUB(q, p)**, P_EXT(x, q, y), **PROP(a, p, b)**, **PROP(c, q, d)**, PROP(s, r, t), CLASS(a), CLASS(b), CLASS(c), CLASS(d), C_SUB(e, c), C_SUB(f, d), <u>C_SUB(c,</u>*

a), C_SUB(d, b), C_EXT(e, x), C_EXT(f, y), p=q=r=“paints”, a=c=s=“Painter”, b=d=t=“Painting”

Using the C_SUB transitivity constraint twice with the homomorphism *{e→f, c→d, a→b}* and the elementary one, we get:

ans(x) :- P_SUB(q, p), P_EXT(x, q, y), PROP(a, p, b), PROP(c, q, d), PROP(s, r, t), CLASS(a), CLASS(b), CLASS(c), CLASS(d), ***C_SUB(e, c)****,* ***C_SUB(f, d)****,* **C_SUB(c, a)**, **C_SUB(d, b)**, C_SUB(e, a), C_SUB(f, b), *C_EXT(e, x), C_EXT(f, y), p=q=r=“paints”, a=c=s=“Painter”, b=d=t=“Painting”*

According to the 2nd basic constraint and the homomorphisms *{d→e, c→c}* and *{d→f, c→d}* we get the query:

ans(x) :- P_SUB(q, p), P_EXT(x, q, y), PROP(a, p, b), PROP(c, q, d), PROP(s, r, t), CLASS(a), CLASS(b), CLASS(c), CLASS(d), CLASS(e), CLASS(f), ***C_SUB(e, c), C_SUB(f, d)****,* C_SUB(c, a), C_SUB(d, b), C_SUB(e, a), C_SUB(f, b), *C_EXT(e, x), C_EXT(f, y), p=q=r=“paints”, a=c=s=“Painter”, b=d=t=“Painting”*

Using twice the universally quantifying constraint for C_SUB and the homomorphisms *{subC→e, clas→c}* and *{subC→f, clas→d}* we get

ans(x) :- P_SUB(q, p), P_EXT(x, q, y), PROP(a, p, b), PROP(c, q, d), PROP(s, r, t), CLASS(a), CLASS(b), CLASS(c), CLASS(d), CLASS(e), CLASS(f), ***C_SUB(e, c), C_SUB(f, d)****,* C_SUB(c, a), C_SUB(d, b), C_SUB(e, a), C_SUB(f, b), *C_EXT(e, x), C_EXT(f, y), p=q=r=“paints”, a=***c=***s=***“Painter”****, e=“Painter”, b=***d=***t=***“Painting”****, f=“Painting”*

Since no more chase step can get applied[12], the query above is the universal plan of the initial query. ■

[12] In reality, the query will be chased with all the existentially quantified constraints for CLASS, PROP, C_SUB, P_SUB. Although this is not shown in the universal plan due to space limitation, all the necessary chase steps for checking containment are illustrated; the chase steps ignored do not alter the result.

4.1.1.4 Complexity

In the general case the end of the chase, as we have said, is not decidable. However, the use of restrictions on the used constraints leads to decidable problems in the complexity class of NP or $\prod_2^p$; the stratified-witness property for disjunction-free constraints belongs to the former class.

Proposition 4.4 ([Deu02]): *Chasing a conjunctive query with a set of disjunction-free DEDs meeting the stratified-witness property results in a query of size*

$$O\left(|Q|^{a^{l+1}}\right)$$

where |Q| is the initial query's size, a is the maximum arity of the predicates used in the relational schema on which the constraints are applied and l is the maximum number of ∃-edges on a path in the constraints chase flow graph. ■

4.1.2 Checking RQL$_{UCQ}$ Query Containment Algorithm

Our algorithm for checking whether an RQL$_{UCQ}$ query is contained in another takes as input the two queries in their SWLF translations and chases them with Δ_{RDF}. Although Δ_{RDF} does not preserve stratified-witness because of the 6th basic and 3rd sub constraints, which introduce a cycle containing an ∃-labelled edge in the chase flow graph, it behaves as if stratified-witness was present. Due to the conditions needed to apply a chase step the two conditions do not allow the introduction of an infinite number of fresh variables, which is the inviolable term for the chase to diverge.

Example 4.9: Let's try to chase the following query with the two constraints just mentioned

ans(q) :- P_SUB(q, p)

It chases with the 6th basic constraint to

ans(q) :- P_SUB(q, p), PROP(a_1, q, b_1), PROP(a_2, p, b_2)

and, then, twice with the 3rd sub constraint to

ans(q) :- P_SUB(q, q), P_SUB(p, p), P_SUB(q, p), PROP(a_1, q, b_1), PROP(a_2, p, b_2)

No more chase step with the two constraints can be applied; the chase ends here. ■

In the example above no value is assigned to any variable. However, nothing would change if one of them or both were equated to values. Therefore, there cannot be an infinite execution of chase steps and the chase with Δ_{RDF} terminates. Thus, by reducing the containment problem to the relational equivalent, we give the following theorem based on [Deu02]:

Theorem 4.1: *Suppose two RQL$_{UCQ}$ queries Q_1, Q_2 translated in SWLF, where $Q_2 = \bigcup_j Q_{2j}$, and a set C of DEDs, namely Δ_{RDF}. Q_1's containment in Q_2 under C ($Q_1 \subseteq_C Q_2$) is decidable. Q_1's chase terminates rendering the universal plan SQ_1: $SQ_1 = \bigcup_i SQ_{1i}$, where SQ_{1i}'s are conjunctive queries. Q_1 is contained in Q_2 for every description base DB iff for each i there is j such that there is a containment mapping from Q_{2j} into SQ_{1i} (i.e. SQ_{1i} is contained in Q_{2j}).* ■

In other words, the above theorem states that an RQL$_{UCQ}$ query Q_1 is contained in an RQL$_{UCQ}$ query Q_2 under a set C of constraints if the universal plan of Q_1, namely SQ_1, is contained in Q_2.

Example 4.10: Let us see again the example of Figure 4-1. Given the same constraints as in the chase example earlier, we want to prove that query

```
Q1:    SELECT    X
       FROM      {X; Painter}paints{Y; Painting};
```

is contained in query

```
Q2:    SELECT    X
       FROM      {X; Painter}creates{Y; Painting};
```

Q_1 translates to:

ans(x) :- PROP(a, p, b), C_SUB(c, a), C_SUB(d, b), C_SUB(e, c), C_SUB(f, d), P_SUB(q, p), C_EXT(e, x), C_EXT(f, y), P_EXT(x, q, y), p="paints", c="Painter", d="Painting"

Then, it chases to the following universal plan SQ_1:

ans(x) :- PROP(a, p, b), PROP(g, q, h), PROP(s, r, t), PROP(u, v, w), CLASS(a), CLASS(b), CLASS(c), CLASS(d), CLASS(e), CLASS(f), CLASS(g), CLASS(h), CLASS(i), CLASS(j), CLASS(s), CLASS(t), C_SUB(e, c), C_SUB(f, d), C_SUB(c, a), C_SUB(d, b), C_SUB(e, a), C_SUB(f, b), C_SUB(g, a), C_SUB(h, b), C_SUB(i, g), C_SUB(j, h), C_SUB(i, a), C_SUB(j, b), C_SUB(k, l), C_SUB(m, n), P_SUB(q, p), P_SUB(o, z), C_EXT(e, x), C_EXT(f, y), C_EXT(i, x), C_EXT(j, y), P_EXT(x, q, y), o=p=q=r="paints", v=z="creates", a=c=e=g=i=s="Painter", u=l="Artist", b=d=f=h=j= t="Painting", w=n="Artifact"

Q_2 translates to:

ans(x) :- PROP(a, p, b), C_SUB(c, a), C_SUB(d, b), C_SUB(e, c), C_SUB(f, d), P_SUB(q, p), C_EXT(e, x), C_EXT(f, y), P_EXT(x, q, y), p="creates", c="Painter", d="Painting"

The containment of Q_1 in Q_2 demands that there is a containment mapping from Q_2 to the single conjunctive query of SQ_1. Such containment exists given the homomorphism $\{p\rightarrow v, a\rightarrow u, b\rightarrow w, c\rightarrow k, d\rightarrow m, e\rightarrow e, f\rightarrow f, q\rightarrow o, p\rightarrow z\}$. ■

In section 3.5.1 we have illustrated how the Datalog facts describing an RDF/S schema are interpreted as constraints. Using these constraints, the set of constraints Δ_{RDF} is defined for every RDF/S schema. In the previous theorem, we have defined the reduction of the RQL query containment problem to the relational equivalent based on this set. In the following example we will show that by the slightest reduction to Δ_{RDF} the algorithm for checking containment is no longer valid. Initially we will employ δ_{RDF} in the containment algorithm, instead of Δ_{RDF}. Then, we will consider the set of constraints that excludes from Δ_{RDF} the existentially quantifying constraints.

Example 4.11: Suppose the queries

```
(Q1)   SELECT    X
       FROM      Artist{X}, Painter{X}
```

and

```
(Q2)   SELECT    X
       FROM      ^Painter{X}
```

It is obvious that Q_2 is contained in Q_1. Additionally, Q_1 is contained in Q_2 since *Painter* is a subclass of *Artist* and has no other subclass but itself. However, the containment of Q_1 in Q_2 cannot be deduced using δ_{RDF} due to the lack of knowledge that *Painter* has no subclass but itself. Moreover, the containment of Q_2 in Q_1 cannot be deduced using Δ_{RDF} without the existentially quantifying constraints because we lack the knowledge that there exists a subclass relationship between *Painter* and *Artist*. ■

The above conclusions propagate to the equivalence and minimisation problems, which are built on top of the containment problem.

4.1.2.1 Complexity

As we have seen, the RQL$_{\text{UCQ}}$ containment problem gets reduced to the containment of unions of conjunctive queries under Δ_{RDF}, which depends on the complexity of chase and the one of checking the containment of first query's universal plan in the second query. Since Δ_{RDF} behaves as if stratified-witness was present, the chase of the first query with it terminates and is at least NP-complete. Moreover, the simple containment check between the first query's universal plan and the second query is $\prod_2^p$ -complete.

Note that the size of the universal plan derives from the chase algorithm. Given the SWLF predicates and Δ_{RDF}, the maximum arity in this case is 3 and the maximum number of ∃-edges on a path in the constraints chase flow graph is 2. This path derives from the constraints used to capture the RDF/S semantics. Therefore, the size of the universal plan is at least $O\left(|Q|^{3^{2+1}}\right) = O\left(|Q|^{27}\right)$, where |Q| is the size of the largest conjunctive query forming the initial union.

4.1.3 RQL$_{\text{CORE}}$ Query Containment

Suppose now the fragment of RQL$_{\text{UCQ}}$ queries that *(a)* is built on class and property patterns whose schema (class and property) variables are assigned to values, and *(b)* does not consider proper interpretations of classes and properties. We will call this fragment *RQL$_{CORE}$*. Although it seems very restrictive, RQL$_{\text{CORE}}$ encompasses a large portion of RQL_{UCQ} queries used in real scenarios asking for data information. Thus, it is mainly oriented towards queries performing data & mixed navigation, but encompasses some elementary schema navigation queries, too.

Definition 4.8: *RQL_{CORE} queries have the form of RQL_{UCQ} queries (see Definition 3.5), with the additional restrictions (a) that the RQL patterns $E_i(\bar{U}_i)$ do not consider proper interpretations and (b) that the equality of all schema variables $\$C_i$, $@P_i$ and $\$D_i$ to constants is implied from the equalities.* ■

The complete list of RQL patterns not involving proper interpretations can be found in Appendix A. We should note here that RQL_{CORE} is similar to the RQL fragment considered in [ACK04]: none of them considers proper interpretations.

Example 4.12: The RQL_{UCQ} query

```
SELECT    X
FROM      Artist{X}
```

or equivalently

```
SELECT    X
FROM      $C{X}
WHERE     $C=Artist
```

is an RQL_{CORE} query. On the contrary, the RQL_{UCQ} query

```
SELECT    X
FROM      ^$C{X}
```

is not an RQL_{CORE} query. ■

By restricting the definition of RQL_{UCQ} query containment to RQL_{CORE} queries we get:

Definition 4.9: *An RQL_{CORE} query Q_1 is contained in an RQL_{CORE} query Q_2 ($Q_1 \subseteq Q_2$) given an RDF description schema DS iff for every description base DB conforming to DS the result of Q_1 is contained in that of Q_2 ($\forall DB\ Q_1(DB) \subseteq Q_2(DB)$).* ■

The algorithm for checking whether an RQL$_{CORE}$ query is contained in another takes as input the two queries in their SWLF translations and δ_{RDF}, instead of Δ_{RDF}; the additional information supplied by Δ_{RDF} is no longer needed. As with Δ_{RDF}, the chase with δ_{RDF} terminates; remember that δ_{RDF} is a subset of Δ_{RDF}. Thus, by reducing the containment problem to the relational equivalent, based on [Deu02], we give the following theorem:

Theorem 4.2: *Suppose two RQL$_{CORE}$ queries Q_1, Q_2 translated in SWLF, where $Q_2 = \bigcup_j Q_{2j}$, and a set C of disjunction-free DEDs, namely δ_{RDF}. Q_1's containment in Q_2 under C ($Q_1 \subseteq_C Q_2$) is decidable. The chase of Q_1 with C terminates rendering the universal plan SQ_1: $SQ_1 = \bigcup_i SQ_{1i}$. Q_1 is contained in Q_2 for every description base DB iff for each i there is a j such that there is a containment mapping from Q_{2j} into SQ_{1i} (i.e. SQ_{1i} is contained in Q_{2j}).* ■

In other words, the above theorem states that an RQL$_{CORE}$ query Q_1 is contained in an RQL$_{CORE}$ query Q_2 under a set C of constraints if the universal plan of Q_1, namely SQ_1, is contained Q_2.

Example 4.13: Take the queries

```
Q1:   SELECT     X
      FROM       Painter{X}, Artist{X}
```

and

```
Q2:   SELECT     X
      FROM       Painter{X}
```

The first one translates to

ans_1(x) :- C_SUB(c, a), C_EXT(c, x), a="Painter", C_SUB(d, b), C_EXT(d, x), b="Artist"

while the second one to

ans_2(x) :- C_SUB(c, a), C_EXT(c, x), a="Painter"

The containment of Q_1 into Q_2 can easily be verified. But, containment of Q_2 into Q_1 is not trivial. Therefore, we chase Q_2 with the existentially quantifying constraint for *C_SUB(Painter, Artist)*.

ans$_2$*(x) :- C_SUB(c, a), C_SUB(d, b), C_EXT(c, x), a="Painter", d="Painter", b="Artist"*

There is no need to illustrate more chase steps. There is a containment mapping from Q_1 to Q_2 that uses the homomorphism $\{c \rightarrow c,\ d \rightarrow c,\ x \rightarrow x,\ b \rightarrow b,\ a \rightarrow a\}$. Thus, Q_2 is contained in Q_1. ■

Example 4.14: Using δ_{RDF} we can, also, confirm the containment of the RQL$_{\text{CORE}}$ query

```
SELECT    X
FROM      {X}@paints
```

into

```
SELECT    X
FROM      {X; Painter}@paints, {X}@creates ■
```

The complexity of the RQL$_{\text{CORE}}$ containment problem depends on the chase and the simple containment check, too. Since δ_{RDF} is a set of disjunction-free dependencies that behaves as if stratified-witness is present, the chase of the first query terminates and is NP-complete; the simple containment check between the first query's universal plan and the second query is $\prod_2^p$ -complete, while the universal plan's size is $O\left(|Q|^{27}\right)$, where |Q| is the size of the largest conjunctive query forming the initial union.

4.1.3.1 Why RQL$_{\text{CORE}}$?

The gain from limiting the expressiveness of RQL$_{\text{UCQ}}$ queries to RQL$_{\text{CORE}}$ is double. First of all, the partial (incomplete) knowledge of the RDF/S schema offered from δ_{RDF} suffixes to solve the containment problem; we do not need complete information, i.e. information stating which are all classes, properties and hierarchies. Additionally, considering only the information provided from δ_{RDF} leads to lower complexity in the containment check (and equivalence and minimisation that will be discussed later), which stems from the fact that the chase considers only disjunction-free constraints; fewer chase steps in order to reach the universal plan and smaller universal plan (no additional conjunctive query introduced in the union).

4.2 RQL_{UCQ} Query Equivalence

Definition 4.10: *An RQL_{UCQ} query Q_1 is equivalent to an RQL_{UCQ} query Q_2 ($Q_1 \equiv Q_2$) given an RDF description schema DS iff for every description base DB conforming to DS the result of Q_1 is equivalent to that of Q_2 ($\forall DB\ Q_1(DB) \equiv Q_2(DB)$).* ■

Having defined containment for RQL_{UCQ} queries and an algorithm to check it, the problem of checking RQL_{UCQ} query equivalence is straightforward. It is known from relational theory that two queries Q_1, Q_2 are considered equivalent ($\equiv$) iff $Q_1 \subseteq Q_2$ and $Q_2 \subseteq Q_1$. This result extends to equivalence under a set C of DEDs ($Q_1 \equiv_C Q_2$ iff $Q_1 \subseteq_C Q_2$ and $Q_2 \subseteq_C Q_1$). Thus, the following definition is educed:

Definition 4.11: *Two RQL_{UCQ} queries Q_1 and Q_2 translated in SWLF are equivalent under a set C of DEDs ($Q_1 \equiv_C Q_2$), namely Δ_{RDF}, iff Q_1 is contained in Q_2 under C ($Q_1 \subseteq_C Q_2$) and vice-versa ($Q_2 \subseteq_C Q_1$).* ■

Example 4.15: We are going to check for equivalence the following queries.

```
(Q1)    SELECT    X
        FROM      {X}paints{Y};
```

and

```
(Q2)    SELECT    X
        FROM      {X; Painter}paints{Y; Painting};
```

At first the queries translate in SWLF. Q_1 translates to

ans(x) :- P_SUB(q, p), P_EXT(x, q, y), p="paints" and, then, chases to SQ_1:

ans(x) :- P_SUB(q, p), P_EXT(x, q, y), PROP(a, p, b), PROP(c, q, d), PROP(s, r, t), CLASS(a), CLASS(b), CLASS(c), CLASS(d), CLASS(e), CLASS(f), C_SUB(e, c), C_SUB(f, d), C_SUB(c, a), C_SUB(d, b), C_SUB(e, a), C_SUB(f, b), C_EXT(e, x), C_EXT(f, y), p=q=r="paints", a=c=e=s="Painter", b=d=f=t="Painting"

Q_2 translates to

ans(x) :- PROP(a, p, b), C_SUB(e, c), C_SUB(f, d), C_SUB(c, a), C_SUB(d, b), P_SUB(q, p), C_EXT(e, x), C_EXT(f, y), P_EXT(x, q, y), p="paints", c ="Painter", d="Painting"

and, then, chases to SQ_2:

ans(x) :- PROP(a, p, b), PROP(g, q, h), PROP(s, r, t), CLASS(a), CLASS(b), CLASS(c), CLASS(d), CLASS(e), CLASS(f), CLASS(g), CLASS(h), CLASS(i), CLASS(j), CLASS(s), CLASS(t), C_SUB(e, c), C_SUB(f, d), C_SUB(c, a), C_SUB(d, b), C_SUB(e, a), C_SUB(f, b), C_SUB(g, a), C_SUB(h, b), C_SUB(i, g), C_SUB(j, h), C_SUB(i, a), C_SUB(j, b), P_SUB(q, p), C_EXT(e, x), C_EXT(f, y), C_EXT(i, x), C_EXT(j, y), P_EXT(x, q, y), p=q=r="paints", a=c=e=g=i=s="Painter", b=d=f=h=j=t="Painting"

There is a containment mapping both from Q_2 to SQ_1, using the elementary homomorphism, and from Q_1 to SQ_2, using the elementary homomorphism, too. Thus, $Q_1 \subseteq_C Q_2$ and $Q_2 \subseteq_C Q_1$ and the queries are equivalent. ■

Example 4.16: Another interesting equivalence test is the following one

```
SELECT    X
FROM      {X; Painter}creates, {X; Sculptor}creates
```

and

```
SELECT    X
FROM      {X; Painter}creates, Sculptor{X}
```

These queries consider an extended RDF/S schema that involves the classes *Sculptor* and *Sculpture* and the property *sculpts* having the previous classes as domain and range, respectively. The following facts are, also, true: *P_SUB(sculpts, creates)*, *C_SUB(Sculptor, Artist)*, *C_SUB(Sculpture, Artifact)*. These facts imply some changes to Δ_{RDF}. The first query translates to

ans(x) :- PROP(a_1, p_1, b_1), P_SUB(q_1, p_1), P_EXT(x, q_1, y_1), C_SUB(c_1, a_1), C_SUB(e_1, c_1), C_EXT(e_1, x), PROP(a_2, p_2, b_2), P_SUB(q_2, p_2), P_EXT(x, q_2, y_2), C_SUB(c_2, a_2), C_SUB(e_2, c_2), C_EXT(e_2, x), p_1=p_2="creates", c_1="Painter", c_2="Sculptor"

The second one translates to

ans(x) :- PROP(a_1, p_1, b_1), P_SUB(q_1, p_1), P_EXT(x, q_1, y_1), C_SUB(c_1, a_1), C_SUB(e_1, c_1), C_EXT(e_1, x), C_SUB(e_2, c_2), C_EXT(e_2, x), p_1="creates", c_1="Painter", c_2="Sculptor"

By chasing them and checking for containment mappings the queries prove to be equivalent. ■

The complexity of the RQL_{UCQ} equivalence problem is the same with the complexity of the RQL_{UCQ} containment problem. Remember that the equivalence check corresponds to minimum one and maximum two containment checks.

4.2.1 RQL_{CORE} Query Equivalence

By paraphrasing the definition of RQL_{UCQ} query equivalence we get the following definition for the case of RQL_{CORE} queries.

Definition 4.12: *An RQL_{CORE} query Q_1 is equivalent to an RQL_{CORE} query Q_2 ($Q_1 \equiv Q_2$) given an RDF description schema DS iff for every description base DB conforming to DS the result of Q_1 is equivalent to that of Q_2 ($\forall DB\ Q_1(DB) \equiv Q_2(DB)$).* ■

Having defined an algorithm to check containment of RQL_{CORE} queries, checking RQL_{CORE} query equivalence is straightforward.

Definition 4.13: *Two RQL_{CORE} queries Q_1 and Q_2 translated in SWLF are equivalent under a set C of EDs ($Q_1 \equiv_C Q_2$), namely δ_{RDF}, iff Q_1 is contained in Q_2 under C ($Q_1 \subseteq_C Q_2$) and vice-versa ($Q_2 \subseteq_C Q_1$).* ■

Obviously, the complexity of the RQL_{CORE} equivalence problem is the same with the one of the RQL_{CORE} containment problem.

4.3 *RQL_{UCQ} Query Minimisation*

We will introduce the problem using an example.

Example 4.17: Suppose the RQL_{UCQ} query

```
SELECT      X, Y
FROM        {X}paints{Y}, {X}creates{Y}
```

needs to be answered. It is pretty obvious that this query leads to redundant processing; the pairs *[x, y]* belonging to the extended interpretation of *paints* belong to the extended interpretation of *creates*, too ($[x, y]_{paints} \subseteq [x, y]_{creates}$). By set theory, $([x, y]_{paints} \cap [x, y]_{creates}) \equiv [x, y]_{paints}$. Thus, this query can minimise to

```
SELECT      X, Y
FROM        {X}paints{Y} ■
```

However, the previous is a simple example and the minimisation is straightforward. There are cases where the size of the query does not allow such deductions. Moreover, we need an automated way for minimising RQL_{UCQ} queries and calculating their minimal equivalents, i.e equivalent queries that are free of redundancy.

Definition 4.14: *Given an RDF description schema DS, an RQL_{UCQ} query Q gets minimised when replaced with a minimal equivalent query SQ ($\forall DB$ $Q(DB) \equiv SQ(DB)$).* ■

A minimal RQL_{UCQ} uses less and/or simpler RQL patterns than the original RQL_{UCQ} query. The basic idea is that a class pattern is simpler than a property pattern; a pattern involving proper interpretations and/or fewer variables is simpler than one involving extended interpretations and/or more variables.

Example 4.18: Suppose the RQL_{UCQ} queries

```
SELECT    X
FROM      ^Painter{X}
```

and

```
SELECT    X
FROM      Painter{X}, Artist{X}
```

The two queries are equivalent. Moreover, the first one is free of redundancy; therefore, it is minimal. ■

The core of the RQL_{UCQ} minimisation is the *backchase algorithm* ([Deu02]), which is used to discover the minimal reformulations.

4.3.1 Backchase Algorithm

The backchase algorithm [Deu02] checks all the subqueries of the universal plan for equivalence to the original query. Thus, the backchase constitutes an application of the chase algorithm for each subquery. The following definitions formulate formally the notion of subqueries and state when a query is minimal according to [Deu02].

Definition 4.15: *A conjunctive query SQ is a subquery of conjunctive query Q if there exists a containment mapping h from SQ into Q such that whenever the image of two distinct atoms R(x) and R(y) under h coincides, the conjunction of equalities x=y is implied by the equality atoms in SQ.*

Definition 4.16: *A union of conjunctive queries* $SQ \leftarrow \bigcup_i SQ_i$ *is a subquery of the union of conjunctive queries* $Q \leftarrow \bigcup_j Q_j$ *if for every i there is a j such that* SQ_i *is a subquery of* Q_j *in the sense of the previous definition.*

Definition 4.17: *Let Q be a union of conjunctive queries* $Q = \bigcup_{1 \le i \le M} Q_i$ *and D be a set of DEDs. We say that Q is D-minimal*[13] *if:*

(i) there are no distinct 1≤k,l≤M such that Q_k *is contained in* Q_l *under D, and*

(ii) there is no m, no distinct relational atoms in Q_m*'s body* $Rj\left(\overline{x_j}\right)$ *(for 1≤j≤k for some k) and no conjunctions of equalities* C_j *(for 1≤j≤k) such that denoting with* $Q_{m,j}$ *the query obtained from* Q_m *by replacing* R_j *with* C_j*, we have that* $\bigcup_{1 \le i \le M, i \ne m} Q_i \cup Q_m$ *is equivalent to* $\bigcup_{1 \le i \le M, i \ne m} Q_i \cup \bigcup_{i \le j \le k} Q_{m,j}$ *under D.*

A query Q is a minimal reformulation of query R under D if it is D-minimal and equivalent to R under D. ■

A union of conjunctive queries may have more than one minimal subqueries; the intuition behind the previous definition is that no redundant data accesses are made from any minimal query. According to [Deu02] the backchase algorithm retrieves all minimal queries.

Theorem 4.3 ([Deu02]): *Given a union of conjunctive queries Q and a set of DEDs C, if the chase of Q under C terminates yielding the universal plan U, all C-minimal reformulations of Q under C are subqueries of U.*

4.3.1.1 Complexity

As with chase, the termination of the backchase is not guaranteed in the general case. It depends on whether the constraints considered guarantee termination of the chase. However, even when this is the case, the backchase is much more expensive from the chase itself. To be exact, a full minimisation where the backchase performs a blind search between candidate subqueries from the universal plan leads to an NP-complete problem in the number of chase sequences, i.e. an exponential number of NP-complete problems!

[13] The definition does not say anything about minimising the number of equality atoms in the minimal query. However, as in [Deu02], throughout this thesis we assume w.l.o.g. that the set of equalities in a minimal query is transitively closed, i.e. it is maximal, even if the transitive closure is not always illustrated in the examples.

In practice, approximate algorithms are used. Such an algorithm is the one adopted to solve the Disjunctive Plan Minimisation (DPM) problem ([Ono05]). This algorithm returns one minimal which is not always the minimum one, i.e. the one with the least number of queries. In order to solve the DPM problem, we reduce it to Set-Cover, where given a universe *U* of n elements and a collection *S* of subsets of *U* ($S=\{S_1, ..., S_k\}$), we search for a minimum cardinality subcollection of *S* that covers all elements of *U*. By employing DPM finding a minimal query is NP-complete.

4.3.2 Minimisation of RQL_{UCQ} Queries

By reducing the RQL_{UCQ} minimisation problem to its relational equivalent we get the following definition:

Definition 4.18: *An RQL_{UCQ} query translated in SWLF gets minimised when replaced with a minimal equivalent under a set of DEDs C, namely Δ_{RDF}, query.* ■

So, our algorithm for minimising an RQL_{UCQ} query takes as input the universal plan of the query we want to minimise, expressed in SWLF, and Δ_{RDF}. Then, all universal plan's subqueries are checked *(i)* for minimality based on Definition 4.14 and *(ii)* equivalence against the universal plan (or the initial query).

Example 4.19: Does the query below minimise? If yes, find a minimal equivalent one.

```
SELECT    X, Y
FROM      {X}paints{Y}, {X; Painter}creates{Y; Painting}
```

The query translates in SWLF to

ans(x, y) :- P_SUB(q_1, p_1), P_EXT(x, q_1, y), p_1="paints", PROP(a_2, p_2, b_2), P_SUB(q_2, p_2), C_SUB(c_2, a_2), C_SUB(d_2, b_2), C_SUB(e_2, c_2), C_SUB(f_2, d_2), C_EXT(e_2, x), C_EXT(f_2, y), P_EXT(x, q_2, y), p_2="paints", c_2="Painter", d_2="Painting"

which chases to

ans(x, y) :- P_SUB(q_1, p_1), P_EXT(x, q_1, y), PROP(a_1, p_1, b_1), PROP(c_1, q_1, d_1), PROP(s_1, r_1, t_1), CLASS(a_1), CLASS(b_1), CLASS(c_1), CLASS(d_1), CLASS(e_1), *CLASS(f_1), C_SUB(e_1, c_1), C_SUB(f_1, d_1), C_SUB(c_1, a_1), C_SUB(d_1, b_1), C_SUB(e_1, a_1), C_SUB(f_1, b_1), C_EXT(e_1, x), C_EXT(f_1, y), p_1=q_1=r_1="paints", a_1=c_1=e_1=s_1="Painter", b_1=d_1=f_1=t_1="Painting", PROP(a_2, p_2, b_2), PROP(g_2, q_2, h_2), PROP(s_2, r_2, t_2), CLASS(a_2), CLASS(b_2), CLASS(c_2), CLASS(d_2), CLASS(e_2), CLASS(f_2), CLASS(g_2), CLASS(h_2), CLASS(i2), CLASS(j_2), CLASS(s_2), CLASS(t_2), C_SUB(c_2, a_2), C_SUB(d_2, b_2), C_SUB(e_2, c_2), C_SUB(f_2, d_2), C_SUB(e_2, a_2), C_SUB(f_2, b_2), C_SUB(g_2, a_2), C_SUB(h_2, b_2), C_SUB(i_2, g_2), C_SUB(j_2, h_2), C_SUB(i_2, a_2), C_SUB(j_2, b_2), P_SUB(q_2, p_2), C_EXT(e_2, x), C_EXT(f_2, y), C_EXT(i_2, x), C_EXT(j_2, y), P_EXT(x, q_2, y), p_2=q_2="creates", c_2=e_2="Painter", a_2=g_2=i_2=s_2="Artist", d_2=f_2="Painting", b_2=h_2=j_2=t_2="Artifact"*

$\cup$ *ans(x, y) :- …, q_2="creates", g_2=i_2= Artist", h_2="Artifact", j_2="Painting"*

$\cup$ *ans(x, y) :- …, q_2="creates", g_2="Artist", i_2="Painter", h_2=j_2="Artifact"*

$\cup$ *ans(x, y) :- …, q_2="creates", g_2="Artist", i_2="Painter", h_2="Artifact", j_2="Painting"*

$\cup$ *ans(x, y) :- …, q_2="paints", g_2=i_2="Painter", h_2=j_2="Painting"*

During backchase we inspect subquery

ans(x, y) :- P_EXT(x, q_1, y), q_1="paints"

which, amazingly, is a minimal reformulation; it is Δ_{RDF}-equivalent to the initial query and no atom can be removed without disturbing equivalence. Additionally, if the translations of RQL patterns are examined, we discover that this minimal query corresponds to the RQL_{UCQ} query

```
SELECT    X, Y
FROM      {X}^paints{Y} ■
```

In the example above the query has only one minimal equivalent. However, this is not always the case. The following example illustrates that.

Example 4.20: Suppose the query

```
SELECT    X
FROM      $C{X; Artist}
```

and the extended RDF/S schema where the class hierarchy graph rooted on class *Artist* is shown in Figure 4-5. The query translates to *ans(x) :- C_SUB(d, c), C_SUB(e, d), C_EXT(e, x), d="Artist"*

This query expressed in terms of SWLF has three (!) minimal equivalents under Δ_{RDF}:

```
(1st)   ans(x) :- C_SUB(e, d), C_EXT(e, x), d="Artist"

(2nd)   ans(x) :- C_EXT(e, x), e="Artist"
        ∪       ans(x) :- C_EXT(e, x), e="Sculptor"
        ∪       ans(x) :-  C_SUB(e, d), C_EXT(e, x), d="Painter"

(3rd)   ans(x) :- C_EXT(e, x), e="Artist"
        ∪       ans(x) :-  C_EXT(e, x), e="Sculptor"
        ∪       ans(x) :-  C_EXT(e, x), e="Painter"
        ∪       ans(x) :-  C_EXT(e, x), e="Cubist"
```

which correspond to the RQL_{UCQ} queries

```
(1st)   SELECT    X
        FROM      Artist{X}

(2nd)   SELECT    X
        FROM      ^Artist{X}

        UNION

        SELECT    X
        FROM      ^Sculptor{X}

        UNION

        SELECT    X
        FROM      Painter{X}
```

```
(3rd)   SELECT    X
        FROM      ^Artist{X}

        UNION

        SELECT    X
        FROM      ^Sculptor{X}

        UNION

        SELECT    X
        FROM      ^Painter{X}

        UNION

        SELECT    X
        FROM      ^Cubist{X}
```

In the first query redundancy has been removed without resolving the navigational part occurring from the traversal of the subclass hierarchy of *Artist*; that is why the extended interpretation of *Artist* is used. On the contrary, in the third minimal query schema information has been unfolded, a union has been introduced and only the proper interpretations of *Artist*'s subclasses are used. The second query lays somewhere in the middle; a part of the schema information has been unfolded, while some other has not. ■

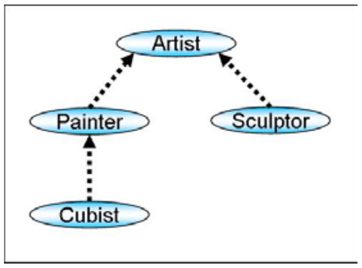


Figure 4-5: Class hierarchy rooted on *Artist* for Example 4.20

Generally, the number of minimal queries depends on the constraints considered and the query given as input. In our RQL_{UCQ} minimisation scenario, Δ_{RDF} describes the classes, the properties and their hierarchies, and the query states which part of the RDF/S schema will be accessed. As the number of constraints and the schema part accessed from the query grow, the number of minimal equivalents considerably increases.

Every RQL_{UCQ} query has one minimal equivalent query where schema information is fully unfolded and no schema navigation needs to take place. Apart from it, there usually exists one minimal query where the unfolding has not introduced union and several ones where partial unfolding has taken place.

The minimal queries that are of interest to us are *(i)* the one where all schema information has been incorporated and *(ii)* the one where unfolding has not introduced union when such a minimal exists. In the aforementioned example, these are the first and third minimal queries, respectively. The former if executed will need to query the RDF/S schema, while the latter will not. In the remaining minimals, if any, some of the conjunctive queries forming each one of them need access to the RDF/S schema and some others do not; that is the case of the second query in the previous example. However, there is usually no reason in picking such a minimal query for execution; only the presence of cached query results could render such queries useful.

An additional advantage of the minimisation procedure is the fact that the RQL_{UCQ} minimal query produced by unfolding all schema information can be used by other query languages, like SPARQL ([PS05]), that consider only the RDF/S data layer, i.e. those having access only to proper interpretations.

Although this example involved a query asking exclusively for data information, a plethora of minimal queries may appear even for RQL$_{UCQ}$ queries not involving class/property interpretations, i.e. schema navigation ones. When the case, the minimal query where schema information is unfolded consists of one or more constant queries, which have the form $ans(x):-C(x)$, where x is a tuple of variables and C is a conjunction of equality atoms between the variables of x and constants. When this is the case, the minimisation algorithm rather answers than minimises schema navigation queries.

Example 4.21: The following query not accessing class/property interpretations

```
SELECT      $D
FROM        Artist{$D}
```

will translate to

ans(d) :- C_SUB(d, c), c="Artist"

which will chase to

ans(d) :- C_SUB(d, c), CLASS(c), CLASS(d), c="Artist", d="Artist"
$\cup$ *ans(d) :- C_SUB(d, c), CLASS(c), CLASS(d), c="Artist", d="Painter"*

This query has two minimal equivalents. The first one is the initial query and the second one is

ans(d) :- d="Artist"
$\cup$ *ans(d) :- d="Painter"* ■

In Example 4.11 we have seen that by relaxing the set Δ_{RDF}, the containment algorithm is no longer valid. We will use the same query in order to illustrate the effects in the minimisation procedure.

Example 4.22: If we use the backchase algorithm with δ_{RDF}, the query

```
SELECT      X
FROM        Painter{X}, Artist{X}
```

will minimise to

```
SELECT      X
FROM        Painter{X}
```

The same query will minimise to

```
SELECT      X
FROM        ^Artist{X}

UNION

SELECT      X
FROM        Artist{Painter}, ^Painter{X}
```

if Δ_{RDF} without the existentially quantifying constraints is considered during backchase. However, when Δ_{RDF} is considered, the minimal query outputted is

```
SELECT      X
FROM        ^Painter{X}
```

It is obvious that Δ_{RDF} is indispensable for a complete minimisation. ■

4.3.2.1 Complexity

The set of constraints Δ_{RDF} guarantees termination of backchase. Therefore, by reducing the RQL$_{UCQ}$ minimisation problem to the equivalent problem of minimising a conjunctive query under Δ_{RDF}, we inherit the complexity of full minimisation.

If we are interested only in the minimal query where all schema information has been unfolded, we can use the following technique. As soon as the chase ends we extract from the universal plan the maximally contained query that uses only the SWLF predicates C_EXT and P_EXT. This query has only one minimal equivalent; thus, we can use the algorithm for solving the DPM problem.

4.3.3 RQL$_{CORE}$ Query Minimisation

Having defined RQL$_{CORE}$, the problem of query minimisation extends to it.

Definition 4.19: *Given an RDF description schema DS, an RQL_{CORE} query Q gets minimised when replaced with an equivalent minimal query SQ ($\forall$DB Q(DB)$\equiv$SQ(DB)).* ■

An RQL_{CORE} minimal query is redundancy-free; it uses less and/or simpler RQL patterns than the original RQL_{CORE} query. The basic idea is that a class pattern is simpler than a property pattern and a pattern involving fewer variables is simpler than one involving more variables.

As with containment of RQL_{CORE} queries, the problem reduces to an equivalent relational one:

Definition 4.20: *An RQL_{CORE} query translated in SWLF gets minimised when replaced with an equivalent minimal query under a set of disjunction-free DEDs C, namely δ_{RDF}.* ■

Once more the backchase is the core of the minimisation process. The RQL_{CORE} queries translated in SWLF get chased with δ_{RDF} and all subqueries of the universal plan are examined for minimality and equivalence to it.

Example 4.23: Suppose we want to minimise the query Q_1

```
SELECT    X
FROM      $C{X}, $E{X}
WHERE     $C=Artist, $E=Painter
```

The query translated in SWLF has the form

ans_1(x) :- C_SUB(d, c), C_EXT(d, x), C_SUB(f, e), C_EXT(f, x), c="Artist", e="Painter"

After being chased with δ_{RDF} the query becomes

ans_1(x) :- C_SUB(d, c), C_SUB(g, d), C_EXT(g, x), C_SUB(f, e), C_SUB(h, f), C_EXT(h, x), c="Artist", e="Painter", C_SUB(h, g), g="Artist", h="Painter"

Suppose now the subquery Q_2

ans_2(x) :- C_SUB(f, e), C_EXT(f, x), e="Painter"

which corresponds to the query

```
SELECT    X
FROM      $E{X}
WHERE     $E=Painter
```

Q_2 is equivalent to Q_1 under δ_{RDF} and is minimal, too. Therefore, Q_1 minimises to Q_2. ■

The complexity of the RQL$_{CORE}$ minimisation problem is the one of full minimision of a conjunctive query under disjunction-free DEDs. However, having observed that RQL$_{CORE}$ queries have only one minimal equivalent, instead of the full minimisation we may adopt Disjunctive Plan Minimisation without losses.

4.3.4 Simplification of RQL Patterns

Earlier in section 4.3, it was stated that RQL$_{UCQ}$ minimal queries contain less and/or simpler patterns than the original queries. It is obvious that a class pattern is simpler than a property pattern. But, can we prove that a pattern is simpler than and equivalent to another under given conditions and how? Our minimisation technique for RQL$_{UCQ}$ queries can be used as a proof procedure for that, too. It allows simplifying RQL patterns in their general form without taking under consideration specific RDF/S schemas; therefore the chase in this case considers only δ_{Mod}.

Example 4.24: Suppose the RQL$_{UCQ}$ query

ans(X, @P, Y) :- {X; $C}@P{Y; $D}, cond(@P, Y)

that involves the pattern we want to simplify and a dummy predicate *cond* stating the conditioned variables. The equivalent query in terms of SWLF is Q_1:

ans(x, p, y) :- PROP(a, p, b), P_SUB(q, p), P_EXT(x, q, y), C_SUB(c, a), C_SUB(d, b), C_EXT(c, x), C_EXT(d, y), cond(p, y)

Chasing with (half of) the 6th basic constraint we get

ans(x, p, y) :- PROP(a, p, b), PROP(e, q, f), P_SUB(q, p), P_EXT(x, q, y), C_SUB(c, a), C_SUB(d, b), C_EXT(c, x), C_EXT(d, y), cond(p, y)

Chasing with the 1st domain-range constraint we get

ans(x, p, y) :- PROP(a, p, b), PROP(m, p, n), PROP(e, q, f), P_SUB(q, p), P_EXT(x, q, y), C_SUB(g, e), C_SUB(h, f), C_SUB(c, a), C_SUB(d, b), C_EXT(g, x), C_EXT(h, y), C_EXT(c, x), C_EXT(d, y), x cond(p, y)

Now, chasing with the 2nd domain-range constraint we get

ans(x, p, y) :- PROP(a, p, b), PROP(m, p, n), PROP(e, q, f), P_SUB(q, p), P_EXT(x, q, y), C_SUB(g, e), C_SUB(h, f), C_SUB(c, a), C_SUB(d, b), C_SUB(e, a), C_SUB(f, b), C_EXT(g, x), C_EXT(h, y), C_EXT(c, x), C_EXT(d, y), cond(p, y)

Then, chasing with C_SUB transitivity we get

ans(x, p, y) :- PROP(a, p, b), PROP(m, p, n), PROP(e, q, f), P_SUB(q, p), P_EXT(x, q, y), C_SUB(g, e), C_SUB(h, f), C_SUB(c, a), C_SUB(d, b), C_SUB(e, a), C_SUB(f, b), C_SUB(g, a), C_SUB(h, b), C_EXT(g, x), C_EXT(h, y), C_EXT(c, x), C_EXT(d, y), cond(p, y)

Finally, by applying the 1st and 3rd basic constraints, the query chases to the universal plan U_1:

ans(x, p, y) :- PROP(a, p, b), P_SUB(q, p), C_SUB(e, a), C_SUB(f, b), PROP(e, q, f), P_EXT(x, q, y), C_SUB(g, e), C_SUB(h, f), C_EXT(g, x), C_EXT(h, y), C_SUB(g, a), C_SUB(h, b), C_SUB(c, a), C_SUB(d, b), C_EXT(c, x), C_EXT(d, y), CLASS(a), CLASS(b), CLASS(c), CLASS(d), CLASS(e), CLASS(f), CLASS(g), CLASS(h), cond(p, y)

Now, during backchase we check the universal plan's subquery Q_2

ans(x, p, y) :- P_SUB(q, p), P_EXT(x, q, y), cond(p, y)

for equivalence to the original query. Thus, we chase it using the same chase steps as for Q_1 and get the universal plan U_2:

ans(x, p, y) :- PROP(a, p, b), P_SUB(q, p), C_SUB(e, a), C_SUB(f, b), PROP(e, q, f), P_EXT(x, q, y), C_SUB(g, e), C_SUB(h, f), C_EXT(g, x), C_EXT(h, y), C_SUB(g, a), C_SUB(h, b), cond(p, y)

We can conclude that Q_2 is equivalent to Q_1. There is a containment mapping from Q_1 to U_2 ($Q_2 \subseteq Q_1$) using the homomorphism $\{x \to x, p \to p, y \to y, a \to a, b \to b, q \to q, c \to g, dh\}$. Moreover, there is a containment mapping from Q_2 to U_1 ($Q_1 \subseteq Q_2$) using the elementary homomorphism.

Now, we examine the RQL property patterns' translations from Appendix B. We can observe that subquery Q_2 corresponds to the SWLF translation of the RQL$_{UCQ}$ query

ans(X, @P, Y) :- {X}@P{Y }, cond(@P, Y)

Thus, the pattern *{X; $C}@P{Y; $D}* gets simplified to pattern *{X}@P{Y}* when only variables *@P*, *Y* are either conditioned or projected. ■

Using the same methodology a number of pattern simplifications can be proven.

Example 4.25: The query

ans(X) :- $C{X; $D}

which corresponds to the SWLF query

ans(x) :- C_SUB(d, c), C_SUB(e, d), C_EXT(e, x)

proves to be equivalent to query

ans(X) :- ^$C{X}

which corresponds to the SWLF query

ans(x) :- C_EXT(e, x)

by applying the same sequence of chase steps as in the previous example. ■

In general RQL patterns have one simplified (minimal) equivalent. Thus, the DPM algorithm can be used instead of full minimisation. The simplified RQL pattern is extracted from the minimal SWLF query[14] by investigating the RQL patterns's translations. Alternatively, the backward translation for minimal RQL_{UCQ} queries, as presented right below (subsection 4.3.5.1), can be used.

4.3.5 Backward Translation to RQL of Minimal Queries

In some cases we would like to restore a query expressed with SWLF terms into an RQL_{UCQ} (RQL_{CORE}) query. As we will see, the translation is rather simple in the case of RQL_{CORE} queries; on the contrary, in the case of RQL_{UCQ} queries it is more complicated.

4.3.5.1 The Case of Minimal RQL_{UCQ} Queries

The translation procedure for minimal SWLF queries corresponding to RQL_{UCQ} queries takes place in two phases. In the first phase simple RQL patterns are identified in the SWLF queries. Then, in the second one, the simple patterns are combined into more complex ones, whenever possible.

In the first phase, the query's FROM clause gets constructed by mapping every atom

- *C_EXT(d, x)*[15] along with an atom *C_SUB(d, c)* – must not exist another atom referencing *d* – to the RQL class pattern *$C{X}*, otherwise to the RQL class pattern *^$D{X}*
- *CLASS(C)* to the RQL class pattern *$C*
- *P_EXT(x, q, y)* along with an atom *P_SUB(q, p)* to the RQL property pattern *{X}@P{Y}*, otherwise to the RQL property pattern *{X}^@Q{Y}*
- *PROP(a, p, b)* to the RQL property pattern *@P* if none of the following stands: *(i)* if *a*, *b* or both appear in C_SUB predicates – *C_SUB(c, a)*, *C_SUB(d, b)* or both –

[14] If a predicate C_SUB(c, c) or P_SUB(p, p) appears in the minimal query, it should be replaced with CLASS(c) or PROP(a, p, b), respectively. The reason is stated in subsection 4.3.5.1. When this is the case, more than one minimal SWLF queries exist, but only one corresponding RQL simplified pattern.

[15] To be exact all C_EXT predicates minus those referring to literal classes are mapped to RQL class patterns. Those involving literal classes cannot generate patterns because there is no such thing as the proper interpretation of literals in RQL's type system.

and *a*, *b*, or both are not equated to constants, then one of the patterns *{$C}@P*, *@P{$D}*, *{$C}@P{$D}* gets added to the WHERE clause, and *(ii)* if *a*, *b*, or both are equated to constants, then the pattern *@P* and the patterns *$A*, *$B*, or both get added, along with the equalities *$A=domain(@P)*, *$B=range(@P)*, or both in the WHERE clause

- *C_SUB(c, a)*, as long as it was not used with an atom PROP or C_EXT, to the RQL class pattern *$C{$D}* if the equality *$C=$D* is not implied, otherwise to class pattern *$C*
- *P_SUB(q, p)*, if it was not used with an atom *P_EXT(x, q, y)*, to the RQL property patterns *@P*, since the equality *q=p* must hold.

The WHERE clause is formed by adding to the equalities originating from the PROP predicates those found in the SWLF query involving variables appearing in the SWLF predicates. Then, the SELECT clause is formed by the projected variables, i.e. the variables found in the query's head. However, if a projected variable is assigned to a value and does not appear in a SWLF predicate, the value is used in the SELECT clause instead of the variable. This will always be the case of *(a)* schema variables that have been assigned to a value during chase, *(b)* schema variables that were assigned to a value in the initial query and *(c)* variables not bound to patterns in the initial query.

Table 4-1: From simple patterns to more complex property ones

Simple Patterns	Complex Patterns
{X}@P, {$C}@P, $C{X}	*{X; $C}@P{Y}*
{X}@P, {$C}@P, ^$C{X}	*{X; ^$C}@P{Y}*
{X}^@P, {$C}@P, $C{X}	*{X; $C}^@P{Y}*
{X}^@P, {$C}@P, ^$C{X}	*{X; ^$C}^@P{Y}*

In the second phase we explore the FROM clause of the query. We try to compose complex patterns using the simple ones identified in the previous phase. Table 4-1 shows how patterns referring to a property's domain can derive. The same procedure is valid for patterns referring to a property's range or both. Moreover, Table 4-2 shows how simple class patterns can be combined into more complex ones.

Finally, we should replace *(a)* multiple occurrences of equated variables with just one variable and *(b)* in the FROM and SELECT clauses as many schema variables as possible with constants by exploiting the equalities.

We believe that the above translation procedure is the best possible and fully exploits the minimal SWLF query, i.e. does not reintroduce redundancy.

Table 4-2: From simple patterns to more complex class ones

Simple Patterns	**Complex Patterns**
$C{$D}, $D{X}, ^$C{X}	*^$C{X; $D}*
$C{$D}, $D{X}	*$C{X; $D}*
$C{$D}, ^$D{X}, ^$C{X}	*^$C{X; ^$D}*
$C{$D}, ^$D{X}	*$C{X; ^$D}*

Example 4.26: Assume the following Datalog query needs to be translated back to RQL_{UCQ}:

ans(x, c, p, y, d) :- PROP(a, p, b), P_SUB(q, p), P_EXT(x, q, y), C_SUB(c, a), C_SUB(d, b), C_SUB(e, c), C_SUB(f, d), C_EXT(e, x), C_EXT(f, y), p="aProp"

In the first phase it will translate to

```
SELECT    X, $C, @P, Y, $D
FROM      {X}@P{Y}, {$C}@P{$D}, $C{X}, $D{Y}
WHERE     @P=aProp
```

In the second phase one it will take the form

```
SELECT    X, $C, @P, Y, $D
FROM      {X ; $C}@P{Y; $D}
WHERE     @P=aProp
```

By incorporating the only available equality in the SELECT and FROM clauses we get the RQL_{UCQ} query

```
SELECT    X, $C, aProp, Y, $D
FROM      {X ; $C}aProp{Y; $D} ■
```

As stated in section 4.3.2, there exists the special case where the body of a minimised SWLF query contains nothing but equalities. This is the case where the initial RQL_{UCQ} query was a schema navigating query or a constant query. Such a query cannot translate into a query of the form SELECT-FROM-WHERE. However, since the answer is already present in the SWLF query, it can simply be translated into a constant query as follows.

Example 4.27: Suppose the RQL_{UCQ} query

```
SELECT    $C, $D
FROM      $C{$D}
WHERE     $C=Artist and $D=Painter
```

It will minimise to query

ans(c, d) :- c="Artist", d="Painter"

which cannot be written as a SELECT-FROM-WHERE RQL_{UCQ} query. Notice that both projected variables in the initial query are schema ones. The minimal query can translate to the RQL_{UCQ} constant query

ans($C, $D) :- $C=Artist, $D=Painter ■

The above translation procedure may raise some additional questions. First of all, why the C_SUB predicates relating a class to itself are interpreted using the pattern *$C*, instead of *$C{$C}*? The answer is simple; for the chase and backchase algorithms, the atoms CLASS(c) and C_SUB(c, c) are equivalent. On the contrary, this is not the case for RQL; *$C{$C}* implies a check for a subclass relationship, while *$C* does not.

Another important issue concerns the use of the RQL functions *domain(@P)* and *range(@P)* not belonging to the RQL_{UCQ} fragment. This is due to the fact that no RQL pattern exist for imposing explicitly a restriction on a property's domain/range. Nevertheless, such a restriction may appear in a minimal query as illustrated by the following example.

Example 4.28: The query

```
SELECT    @P
FROM      {$C}@P
WHERE     $C=Artist
```

translates to

ans(p) :- C_SUB(c, a), PROP(a, p, b), c="Artist"

If two more properties are defined on *Artist* apart from *creates*, then the query has two minimal equivalents. The query

```
ans(p) :- p="creates"
∪      ans(p) :- p="…"
∪      ans(p) :- p="…"
```

and the query

ans(p) :- PROP(a, p, b), a="Artist"

The first one cannot translate into a SELECT-FROM-WHERE query. The second one can translate only with the "help" of function *domain* to

```
SELECT    @P
FROM      @P, $A
WHERE     $A=domain(@P) and $A=Artist
```

or even better, by exploiting the constants, to

```
SELECT    @P
FROM      @P
WHERE     Artist=domain(@P)
```

4.3.5.2 The Case of Minimal RQL_{CORE} Queries

The query's FROM clause gets constructed by mapping every atom

- *C_EXT(d, x)* along with an atom *C_SUB(d, c)* to the RQL class pattern *$C{X}*

- *P_EXT(x, q, y)* along with an atom *P_SUB(q, p)* to the RQL property pattern *{X}@P{Y}*

The WHERE clause is formed from the equalities involving variables appearing in the SWLF predicates. Finally, the SELECT clause is formed by the projected variables, i.e. the variables found in the query's head. However, if a projected variable not appear in an SWLF predicate is assigned to a value, the value is used in the SELECT clause instead of the variable. This will always be the case of *(a)* schema variables that were assigned to a value in the initial query and *(b)* variables not bound to patterns in the initial query. Finally, we should replace *(a)* all multiple occurrences of equated variables with just one and *(b)* in the FROM and SELECT clauses as many schema variables as possible by exploiting the equalities.

As with RQL_{UCQ} queries, there exists the possibility that a minimal query cannot translate into the SELECT-FROM-WHERE formalism. Once again, to address this issue we follow a similar to RQL_{UCQ} translation approach.

Example 4.29: Let's see the translation of a simple minimal SWLF query corresponding to an RQL_{CORE} one.

The redundant RQL_{CORE} query

```
SELECT    $C, X
FROM      Artist{X; $C}
WHERE     $C=Artist
```

after being translated in SWLF, minimises to

ans(c, x) :- C_SUB(d, c), C_EXT(d, x), c="Artist"

Using the aforementioned algorithm it will translate to the RQL_{CORE} query

```
SELECT    $C, X
FROM      $C{X}
WHERE     $C=Artist
```

or even better to

```
SELECT    Artist, X
FROM      Artist{X}
```

after we incorporate the equalities in the FROM and SELECT clauses. ■

Chapter 5

RQL Query Reformulation

In the previous chapter we have focused on the RQL containment and minimisation problems exploring the potentials of the chase and backchase algorithms. However, these algorithms can, also, be used for reformulating RQL queries given a set of mapping rules from one schema to another.

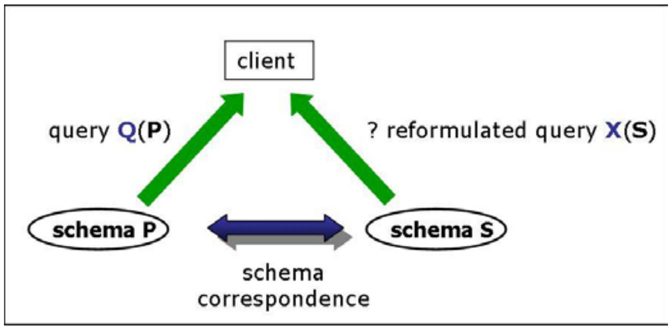


Figure 5-1: The general query reformulation problem

Definition 5.1: *The problem of query reformulation consists of finding (whenever it exists) a query (or queries) Q_S over a schema S that return(s) the same answer to a given query Q_P over a schema P.* ■

The general query reformulation problem is depicted in Figure 5-1. In our case the goal is to express the given RQL_{UCQ} query translated in terms of SWLF into an equivalent query expressed in terms of the underlying relational schema while minimising the output (relational-reformulated) query. In order to succeed in both goals, the following steps are needed. Initially the RQL_{UCQ} query gets rewritten in terms of C_EXT and P_EXT; after that it gets refined and reformulated against the relational schema using RDB→RDF mappings, and minimised using constraints from the RDB. Then, the resulting minimal relational queries are translated into appropriate SQL queries. Once the translation is complete, they can get executed and have their results translated into RDF/S data. Before proceeding to the algorithm's descriptions, the RDB→RDF mappings used to bridge the gap between the relational world and the world of RDF have to be introduced.

5.1 From RDBs Schemas to SWLF

Passing from the relational world to the virtual RDF world requires a mean to associate the two worlds. This is why RDB→RDF mappings are used; these mappings have the form of Datalog rules and need be specified just once for every relational schema. Their goal is to virtually populate the C_EXT and P_EXT relations.

Definition 5.2: *An RDB to RDF mapping is a Datalog rule of the form*

$$\overline{\phi}_{RVLClause}\left(\overline{x}\right) :- \overline{\phi}_{RelationalClause}\left(\overline{x}, \overline{y}\right)$$

where $\overline{\phi}_{RVLClause}\left(\overline{x}\right)$ *is a conjunction of RVL clauses of the form* $A(x_1)$ *or* $A(x_1, x_2)$, *depending on whether each clause refers to the proper extent of an RDF/S class or property, and* $\overline{\phi}_{RelationalClause}\left(\overline{x}, \overline{y}\right)$ *is a conjunction of relational atoms of the form* $R(\omega_1, \ldots, \omega_l)$ *and equality atoms of the form* $\omega=\omega'$, *where* $\omega_1, \ldots, \omega_l, \omega, \omega'$ *are variables or constants.* ■

Artist(x) and *Creates(x, y)* constitute examples of RVL clauses. The translation of the mapping rules in the internal SWLF representation is achieved using the predicate C_EXT for RVL clauses referring to proper extents of classes and the predicate P_EXT for RVL clauses referring to proper extents of properties. The aforesaid RVL clauses would translate into *C_EXT(Artist, x)* and *P_EXT(x, creates, y)*, respectively.

There is one exception to the rule above. Due to their nature, the classes modelling literal types demand special handling; there is no way to limit the possible literal values because they are infinite. Based on this fact, no mapping can be specified for the literal classes.

Example 5.1: Figure 5-2 presents a simple relational schema consisting of four relations whose names illustrate their intended usage. The relation *Painter* provides additional information to the one supplied from *Artist*. Similarly, the relation *Painting* works as complement to *Artifact*. Based on the last definition, Figure 5-3 shows the mappings from the relational schema to the virtual RDF/S schema used in all examples till now. ■

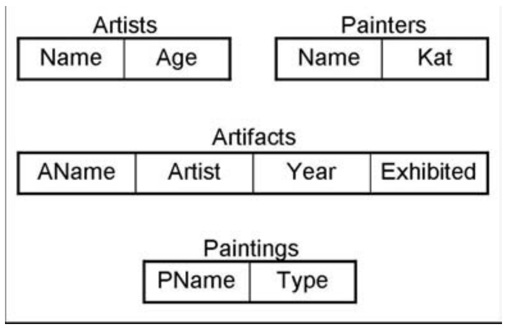


Figure 5-2: Relation database schema

The mappings supported in SWIM are very expressive and partially support the GLAV ([FLM99]) approach, which is a mixed GAV ([Ull00]) and LAV ([Lev99] [Lev01]) approach. More thorough analysis of the expressive power of our mappings can be found in the related work presented in [Kof05].

5.1.1 Translating the Mappings into Constraints

As has been stressed throughout this thesis, all information (RDF/S semantics, RDF/S schema, RDB→RDF mappings) must be given as input to the chase and backchase algorithms in the form of constraints. Hence, the mappings need to get translated into constraints. However, we have to make sure that no information will be lost during translation.

We will start by indicating how constraints are extracted from mappings that follow the GAV approach, i.e. mappings that describe the global RDF/S schema in terms of the local relational one. These mappings have the form

$$\phi_{RVLClause}\left(\overline{x}\right) :- \overline{\phi}_{RelationalClause}\left(\overline{x}, \overline{y}\right)$$

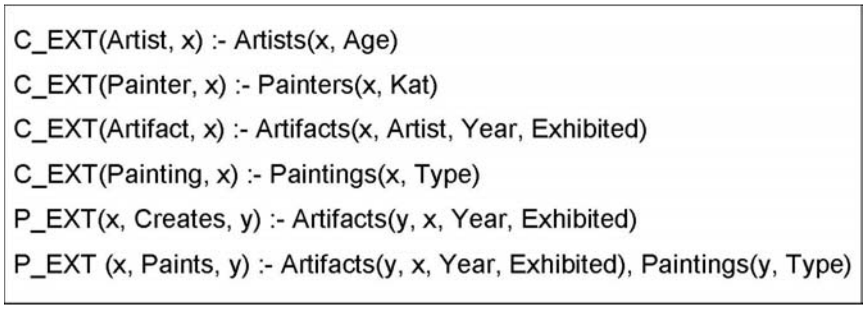
Definition 5.3: *A mapping's interpretation consists of two constraints:*

$$\forall \bar{x} \forall \bar{y} \phi_{RelationalClause}\left(\bar{x}, \bar{y}\right) \rightarrow \phi_{RVLClause}\left(\bar{x}\right)$$

and

$$\forall \bar{x} \phi_{RVLClause}\left(\bar{x}\right) \rightarrow \exists \bar{y} \phi_{RelationalClause}\left(\bar{x}, \bar{y}\right) \blacksquare$$

Both constraints are needed because the first one guarantees soundness of the interpretation and the second one guarantees completeness.



```
C_EXT(Artist, x) :- Artists(x, Age)
C_EXT(Painter, x) :- Painters(x, Kat)
C_EXT(Artifact, x) :- Artifacts(x, Artist, Year, Exhibited)
C_EXT(Painting, x) :- Paintings(x, Type)
P_EXT(x, Creates, y) :- Artifacts(y, x, Year, Exhibited)
P_EXT (x, Paints, y) :- Artifacts(y, x, Year, Exhibited), Paintings(y, Type)
```

Figure 5-3: RDB→RDF mapping rules in SWLF

Example 5.2: Figure 5-4 shows how, using the above definition, the mappings of Figure 5-3 translate into constraints. ■

When the mappings follow the LAV approach, i.e. describe the local relational schema in terms of the global RDF/S one, or GLAV approach, apart from the two constraints presented in Definition 5.3, we can extract one constraint from each RVL clause of the head. These additional constraints can provide valuable information when the mappings are not complete, i.e. when they do not provide information for all class and property interpretations.

Definition 5.4: *From every GLAV (or LAV) mapping we extract the constraints*

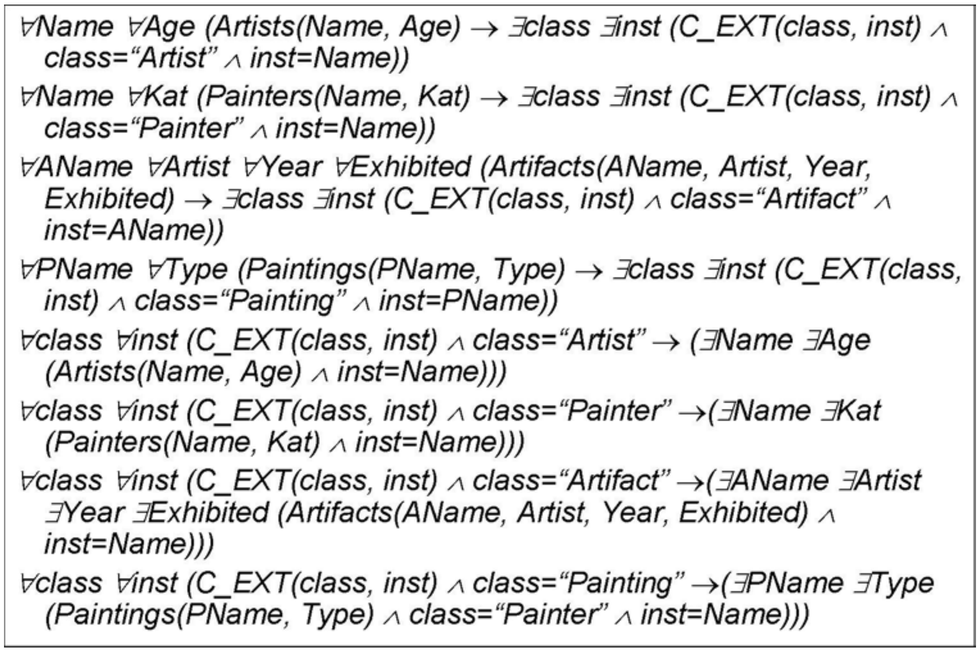
$$\forall \bar{x} \forall \bar{y} \phi_{RelationalClause}\left(\bar{x}, \bar{y}\right) \rightarrow \bar{\phi}_{RVLClause}\left(\bar{x}\right)$$

and

$$\forall \overline{x}\,\overline{\phi}_{RVLClause}\left(\overline{x}\right) \rightarrow \exists \overline{y}\,\overline{\phi}_{RelationalClause}\left(\overline{x},\overline{y}\right)$$

Additionally, for each RVL clause in their body we extract the constraint

$$\forall \overline{x_i}\,\phi^{i}_{RVLClause}\left(\overline{x_i}\right) \rightarrow \exists \overline{x}' \exists \overline{y}\,\overline{\phi}_{RelationalClause}\left(\overline{x},\overline{y}\right), \overline{x}' = \overline{x} - \overline{x_i} \quad \blacksquare$$



∀Name ∀Age (Artists(Name, Age) → ∃class ∃inst (C_EXT(class, inst) ∧ class="Artist" ∧ inst=Name))
∀Name ∀Kat (Painters(Name, Kat) → ∃class ∃inst (C_EXT(class, inst) ∧ class="Painter" ∧ inst=Name))
∀AName ∀Artist ∀Year ∀Exhibited (Artifacts(AName, Artist, Year, Exhibited) → ∃class ∃inst (C_EXT(class, inst) ∧ class="Artifact" ∧ inst=AName))
∀PName ∀Type (Paintings(PName, Type) → ∃class ∃inst (C_EXT(class, inst) ∧ class="Painting" ∧ inst=PName))
∀class ∀inst (C_EXT(class, inst) ∧ class="Artist" → (∃Name ∃Age (Artists(Name, Age) ∧ inst=Name)))
∀class ∀inst (C_EXT(class, inst) ∧ class="Painter" →(∃Name ∃Kat (Painters(Name, Kat) ∧ inst=Name)))
∀class ∀inst (C_EXT(class, inst) ∧ class="Artifact" →(∃AName ∃Artist ∃Year ∃Exhibited (Artifacts(AName, Artist, Year, Exhibited) ∧ inst=Name)))
∀class ∀inst (C_EXT(class, inst) ∧ class="Painting" →(∃PName ∃Type (Paintings(PName, Type) ∧ class="Painter" ∧ inst=Name)))

Figure 5-4: The constraints corresponding to the C_EXT mappings of Figure 5-3

Example 5.3: Take for example the mapping rule

creates(x, y), exhibited(y, z) :- Exhibits(z, y, ...), Artifact(y, x, ...)

In SWLF it will translate into

P_EXT(x, creates, y), P_EXT(y, exhibited, z) :- Exhibits(z, y, ...), Artifact(y, x, ...)

Then it will get interpreted by the constraints

(d_1) ∀x ∀y ∀z ∀p ∀q P_EXT(x, p, y) ∧ P_EXT(y, q, z) ∧ p="creates" ∧ q="exhibited" → ∃... Exhibits(z, y, ...) ∧ Artifact(y, x, …)

(d_2) ∀x ∀y ∀z ∀... Exhibits(z, y, ...) ∧ Artifact(y, x, …) → ∃p∃q P_EXT(x, p, y) ∧ P_EXT(y, q, z) ∧ p="creates" ∧ q="exhibited"

The interesting part is that we can, also, infer the constraints

(d_{11}) $\forall x \forall y \forall p$ $P\_EXT(x, p, y) \wedge p=\text{"creates"} \rightarrow \exists z \exists ...$ $Exhibits(z, y, ...) \wedge Artifact(y, x, ...)$

(d_{12}) $\forall y \forall z \forall q$ $P\_EXT(y, q, z) \wedge q=\text{"exhibits"} \rightarrow \exists x \exists ...$ $Exhibits(z, y, ...) \wedge Artifact(y, x, ...)$

The inverse constraints of (d_{11}) and (d_{12}) are not needed due to the existence of constraint (d_2). ■

5.1.2 Datalog Semantics vs. Constraints Semantics

Our reformulation scenario is initially described by a Datalog program[16] (facts and rules) and a set of constraints. The mapping rules having the same head imply the existence of union when composed with the query. However, as soon as the mapping rules get interpreted as constraints this semantics is lost; constraints having the same body imply a conjunction when to be applied. Therefore, a disjunction has to be stated explicitly by merging the constraints having the same RVL clauses in their body. Notice that this does not happen for the reverse constraints, i.e. those having the same relational atoms in their body; constraints are not merged because disjunction is not implied from the rules' semantics.

Example 5.4: The mapping rules below have the same head

C_EXT(Artist, x) :- Painters(x, …)
C_EXT(Artist, x) :- Sculptors(x, …)

They imply that instances of class *Artist* are given either from the relation *Painters* or from the relation *Sculptors*. Therefore, the constraints

$\forall c \forall x$ $C\_EXT(c, x) \wedge c=\text{"Artist"} \rightarrow \exists ...$ $Painters(x, ...)$

$\forall c \forall x$ $C\_EXT(c, x) \wedge c=\text{"Artist"} \rightarrow \exists ...$ $Sculptors(x, ...)$

[16] To be more precise our reformulation scenario is described by Datalog program when the mappings follow the GAV approach. In the case of GLAV mappings, the terms "Datalog program" and "Datalog rule" are used by misappropriation just to highlight the functionality of SWIM.

merge to constraint

$\forall c \forall x\ C\_EXT(c, x) \wedge c=\text{"Artist"} \rightarrow (\exists ...\ Painters(x, ...) \vee \exists ...\ Sculptors(x, ...))$ ■

Example 5.5: On the contrary, we do not merge the constraints

$\forall y \forall x \forall ...\ Artifacts(y, x, ...) \rightarrow \exists c\ C\_EXT(c, y) \wedge c=\text{"Artifact"}$

$\forall y \forall x \forall ...\ Artifacts(y, x, ...) \rightarrow \exists p\ P\_EXT(x, p, y) \wedge p=\text{"Creates"}$

extracted from the rules

C_EXT(Artifact, y) :- Artifacts(y, x, …)
P_EXT(x, Creates, y) :- Artifacts(y, x, …) ■

5.1.3 Using Functions

A very useful feature of SWIM is that it supports the use of simple functions in the mapping rules, i.e., function names can be used as relational atoms, too. The expressive power of SQL engines provides the only limitation to functions; since the queries, after being reformulated, must be transformed to equivalent SQL queries, the SQL engine must support the functions used at the middleware layer.

The most commonly used function is *Concat(a, b, c)*, which states that the value of variable *a* is given by concatenating the values of variables *b* and *c*. A straightforward use for *Concat* is creating unique URIs for the RDF/S data-result of the RQL_{UCQ} queries. Notice that the values of relations *(a)* are not unique across all relations and *(b)* have not the form of a URI. Thus, if they are used, the resulting RDF/S data will not be valid (except in the case of literal values). The following simple example illustrates how *Concat* can be used to overcome this problem.

Example 5.6: Take the mapping rule populating the *Artist* class in Figure 5-3. This rule could become

C_EXT(Artist, x) :- Concat(x, “http://www.csd.uoc.gr/.../Artist.rdf#”, y), Artists(y, Age)

The result of this rule is creating unique URIs for all *Artist* instances. ■

Normally, all the mappings in the examples used throughout this thesis should make use of the *Concat* function in order to create valid RDF data. However, for simplicity reasons, *Concat* is not used in any mapping in the rest of this thesis.

5.2 Reformulation Phases

The process of reformulating an RQL_{UCQ} query into an SQL one and extracting the RDF/S results takes place in five successive phases. During the first phase the RQL_{UCQ} query gets rewritten against the C_EXT and P_EXT predicates. Then, the query gets reformulated against the relational schema, while, at the same time, some of the conjunctive queries forming the query are removed, because they cannot get reformulated. The third phase minimises the query either by considering constraints from the relational database or not. The forth phase takes the minimal Datalog query and translates it into an equivalent SQL query, while in the last phase the SQL query gets executed and, using its results, RDF/S data get created.

5.2.1 First Phase

The first phase of the RQL_{UCQ} reformulation algorithm should be familiar by now, because it involves the same steps as described for the containment and minimisation problems.

5.2.1.1 Queries Not Involving Class/Property Interpretations

When an RQL_{UCQ} query is a constant one or explores the class and property taxonomy of the virtual RDF/S schema, our approach is based solely on the chase algorithm. This is the case of queries built solely on schema patterns, i.e. those not involving class/property interpretations. Such queries cannot get reformulated against the relational schema. Moreover, all information needed to answer them is present in the universal plans as soon as the chase with Δ_{RDF} ends; there is no reason to apply the backchase algorithm.

Example 5.7: The query

SELECT $D, $C

FROM $C{$D}
WHERE $C=Artist

asks for the subclasses of *Artist*. It can be seen in a rule-based formalism

```
<RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <rdf:Bag>
    <rdf:li>
      <rdf:Seq>
        <rdf:li rdf:type="class" rdf:resource="www.csd.uoc.gr/.../culture.rdf#Painter"/>
        <rdf:li rdf:type="class" rdf:resource="www.csd.uoc.gr/.../culture.rdf#Artist"/>
      </rdf:Seq>
    </rdf:li>
    <rdf:li>
      <rdf:Seq>
        <rdf:li rdf:type="class" rdf:resource="www.csd.uoc.gr/.../culture.rdf#Artist"/>
        <rdf:li rdf:type="class" rdf:resource="www.csd.uoc.gr/.../culture.rdf#Artist"/>
      </rdf:Seq>
    </rdf:li>
  </rdf:Bag>
</RDF>
```

Figure 5-5: RDF/S Data for Example 5.7

ans($D, $C) :- $C{$D}, $C = Artist

which translates in SWLF to

ans(d, c) :- C_SUB(d, c), c = "Artist"

The query chases to the universal plan

ans(d, c) :- C_SUB(d, c), CLASS(c), CLASS(d), c = "Artist", d = "Artist"
$\cup$ *ans(d, c) :- C_SUB(d, c), CLASS(c), CLASS(d), c = "Artist", d = "Painter"*

It is easy to conclude that *(d, c)* $\in$ *{(Painter, Artist), (Artist, Artist)}* is the result. ■

The example above clearly illustrates that the results can be extracted from the universal plan using a simple deductive engine. Moreover, having extracted the results, it is easy to derive RDF/S data from it. Figure 5-5 shows what these data would look like.

Obviously, the reformulation inherits chase's complexity under Δ_{RDF}.

5.2.1.2 Queries Involving Class/Property Interpretations

When an RQL_{UCQ} query navigates both through the virtual RDF/S schema and data, the algorithm considered is familiar; the query translates into SWLF, gets chased with Δ_{RDF} and, then, gets backchased.

Comparing to the minimisation algorithm, a slight modification gets adopted. We are interested only on minimal queries involving nothing but C_EXT and P_EXT predicates; only these queries can be reformulated to queries against the underlying relational schema using the RDB→RDF mappings. Therefore, we use the following technique. As soon as the chase ends, we find the maximal subquery expressed only in terms of C_EXT and P_EXT. This query is guaranteed to be equivalent to the universal plan. Now, we use the backchase on this maximal subquery in order to find the minimal query expressed in terms of C_EXT and P_EXT predicates.

Example 5.8: Suppose the query

SELECT Y
FROM {X; Painter}creates{Y; Painting}

It translates to

ans(y) :- PROP(a, p, b), C_SUB(c, a), C_SUB(d, b), C_SUB(e, c), C_SUB(f, d), P_SUB(q, p), C_EXT(e, x), C_EXT(f, y), P_EXT(x, q, y), p="Creates", c="Painter", d="Painting"

and chases to

ans(y) :- PROP(a, p, b), PROP(g, q, h), C_SUB(c, a), C_SUB(d, b), C_SUB(e, c), C_SUB(f, d), C_SUB(e, a), C_SUB(f, b), C_SUB(g, a), C_SUB(h, b), C_SUB(i, g), C_SUB(j, h), C_SUB(i, a), C_SUB(j, b), P_SUB(q, p), C_EXT(e, x), C_EXT(f, y), *C_EXT(i, x), C_EXT(j, y), P_EXT(x, q, y), p=q="creates", c=e="Painter", a=g=i="Artist", d=f="Painting", b=h=j="Artifact"*

$\cup$ *…, q="creates", g=i="Artist", h="Artifact", j="Painting"*

$\cup$ *…, q="creates", g="Artist", i="Painter", h=j="Artifact"*

$\cup$ *…, q="creates", g="Artist", i="Painter", h="Artifact", j="Painting"*

$\cup$ *…, q="paints", g=i="Painter", h=j="Painting"*

The minimal query involving only C_EXT and P_EXT predicates is

ans(y) :- P_EXT(x, p, y), C_EXT(e, x), C_EXT(f, y), p="creates", e="Painter", f="Painting"

$\cup$ *ans(y) :- P_EXT(x, p, y), p="paints"*

which corresponds to the RQL_{UCQ} query

```
SELECT    Y
FROM      {X; ^Painter}^creates{Y; ^Painting}

UNION

SELECT    Y
FROM      {X}^paints{Y} ■
```

Earlier we have stated that the chase with Δ_{RDF} terminates. Normally, the first phase of the reformulation would inherit the complexity of full minimisation. However, based on the observation that the maximal subquery expressed in terms of C_EXT and P_EXT has only one minimal equivalent, the Disjunction Plan Minimisation can be used.

5.2.2 Second Phase

The RQL_{UCQ} query given as input has been reformulated by now into a minimal query *MQ* expressed in terms of RVL clauses translated into SWLF, i.e. C_EXT and P_EXT. The algorithm's second phase can be seen as an intermediate auxiliary phase. All conjunctive queries MQ_i forming *MQ* get examined one by one in order to locate those that cannot be rewritten into equivalent relational ones. Keep in mind that there is no guarantee that a relational database contains all information corresponding to a specific virtual RDF/S schema.

Therefore, the conjunctive queries containing predicates for which there is no RDB→RDF mapping have to be removed. The procedure, as illustrated below, is rather simple.

$$MQ = \bigcup_i MQ_i, MQ_i \xrightarrow{ChaseWith\Delta_{Map}} MU_i \xrightarrow{Maximal\,RelationalSubqueryExtraction} MR_i$$

Each conjunctive query gets chased with the constraints extracted from the RDB→RDF mappings (*Δ_{Map}*) producing a (union of) conjunctive query(ies) *MU_i* expressed in terms of SWLF and of the underlying relational schema. Then, the maximal subquery *MR_i* expressed in terms of the relational schema gets extracted from each query *MU_i* produced. The initial query *MQ_i* and *MR_i* get tested for equivalence under *Δ_{Map}*; whenever the check fails, the query *MR_i* gets rejected. The remaining *MR_i*'s get combined and form the union *MR*; *MR* is a (union of) conjunctive query(ies) expressed in terms of the relational schema.

Example 5.9: The RQL_{UCQ} query

```
SELECT    X
FROM      Artist{X}
```

after chase and backchase (1st phase) becomes

ans(x) :- C_EXT(Artist, x)
∪ ans(x) :- C_EXT(Painter, x)

Suppose there is no mapping for *C_EXT(Painter, x)*. Thus, the corresponding subquery has to be removed and the query that will be passed to the next phase is

ans(x) :- Artists(x, Age) ■

As we have already mentioned many times, literals are dealt as classes internally to SWIM. Therefore, there will be some limited cases where the minimal query produced from the first phase will contain conjunctive queries built on C_EXT predicates for literals; these queries have to be removed, too, since there are no mapping rules for them. The aforementioned procedure succeeds in removing these queries, too.

Example 5.10: The RQL$_{UCQ}$ query

```
SELECT    X
FROM      $C{X}
```

after chase and backchase (1st phase) becomes

ans(x) :- C_EXT(Artist, x)
∪ *ans(x) :- C_EXT(Painter, x)*
∪ …
∪ *ans(x) :- C_EXT(String, x)*
∪ *ans(x) :- C_EXT(Integer, x)*
∪ …

The removal of the conjunctive queries containing C_EXT predicates that refer to literals leads to the following query expressed in terms of the relational schema:

ans(x) :- Artists(x, Age)
∪ *ans(x) :- Painters(x, Kat)*
… ■

Unfortunately, there is the possibility that all the conjunctive queries MR_i have to be removed. This is the case where the underlying database cannot answer the given RQL$_{UCQ}$ query; therefore, the reformulation procedure ends without returning any result.

The termination of this phase depends once more on the constraints considered. These are the constraints extracted from the mappings (Δ_{Map}). Although they do not satisfy the stratified-witness property, they guarantee termination of the chase[17].

5.2.3 Third Phase

The third phase accepts as input the query produced from the second phase of the reformulation procedure. Although the query is minimal when outputted from the first phase, the rewriting against the relational schema may introduce some redundancy,

[17] Appendix II provides some thoughts and examples that support this conclusion.

depending on the mappings. In order to remove this redundancy, the query has to get backchased.

Example 5.11: The query

```
SELECT    Y
FROM      {X; Painter}creates{Y; Painting}
```

after the 1st phase minimises to

```
SELECT    Y
FROM      {X}^creates{Y}, ^Painter{X}, ^Painting{Y}

UNION

SELECT    Y
FROM      {X}^paints{Y}
```

which in SWLF corresponds to

ans(y) :- P_EXT(x, p, y), C_EXT(e, x), C_EXT(f, y), p="creates", e="Painter", f="Painting"

$\cup$ *ans(y) :- P_EXT(x, p, y), p="paints"*

In the 2nd phase, this query chases with Δ_{Map}, as presented in section 5.1, rendering the universal plan

ans(y) :- P_EXT(x, p, y), C_EXT(e, x), C_EXT(f, y), Artifacts(y, x, Year, Exhibited), Painters(x, Kat), Paintings(y, Type), p="creates", e="Painter", f="Painting"

$\cup$ *ans(y) :- P_EXT(x, p, y), Artifacts(y, x, Year, Exhibited), Paintings(y, Type), p="paints"*

From the universal plan the maximal subquery expressed in terms of the relational schema gets extracted and outputted.

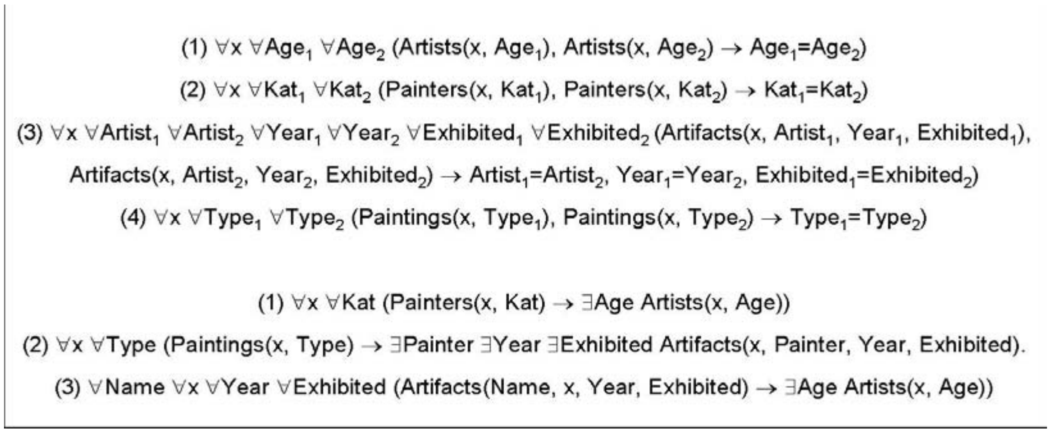
ans(y) :- Artifacts(y, x, Year, Exhibited), Painters(x, Kat), Paintings(y, Type)

$\cup$ *ans(y) :- Artifacts(y, x, Year, Exhibited), Paintings(y, Type)*

The current reformulation phase will use as input the above query and backchase it. As soon as the backchase ends, it will output the minimal query *ans(y) :- Artifacts(y, x, Year, Exhibited), Paintings(y, Type)* ■

5.2.3.1 Exploiting Additional RDB Information in the Minimisation

The end of this phase finds us with a set of minimal queries that will be translated into SQL. However, further minimisation can take place if additional information supplied from the relational database is considered. This information usually comes in the form of integrity constraints and constraints capturing materialised views, if there are any defined. The most commonly used integrity constraints are functional dependencies, like keys, and inclusion dependencies, like foreign keys, which are EDs according to [AHV95]. The (inclusion) constraints interpreting a materialised view are generated in the same way constraints are produced for the mapping rules that follow the GAV approach; one constraint ensures the soundness of the interpretation and a second one the completeness.



(1) $\forall x \, \forall Age_1 \, \forall Age_2$ (Artists(x, Age_1), Artists(x, Age_2) $\rightarrow$ Age_1=Age_2)

(2) $\forall x \, \forall Kat_1 \, \forall Kat_2$ (Painters(x, Kat_1), Painters(x, Kat_2) $\rightarrow$ Kat_1=Kat_2)

(3) $\forall x \, \forall Artist_1 \, \forall Artist_2 \, \forall Year_1 \, \forall Year_2 \, \forall Exhibited_1 \, \forall Exhibited_2$ (Artifacts(x, $Artist_1$, $Year_1$, $Exhibited_1$), Artifacts(x, $Artist_2$, $Year_2$, $Exhibited_2$) $\rightarrow$ $Artist_1$=$Artist_2$, $Year_1$=$Year_2$, $Exhibited_1$=$Exhibited_2$)

(4) $\forall x \, \forall Type_1 \, \forall Type_2$ (Paintings(x, $Type_1$), Paintings(x, $Type_2$) $\rightarrow$ $Type_1$=$Type_2$)

(1) $\forall x \, \forall Kat$ (Painters(x, Kat) $\rightarrow$ $\exists Age$ Artists(x, Age))

(2) $\forall x \, \forall Type$ (Paintings(x, Type) $\rightarrow$ $\exists Painter \, \exists Year \, \exists Exhibited$ Artifacts(x, Painter, Year, Exhibited).

(3) $\forall Name \, \forall x \, \forall Year \, \forall Exhibited$ (Artifacts(Name, x, Year, Exhibited) $\rightarrow$ $\exists Age$ Artists(x, Age))

Figure 5-6: Integrity constraints for the relational schema

In order to exploit such additional information, the third phase gets refined. The maximal rewriting outputted from the 2nd phase gets chased with the relational constraints (Δ_{Rel}) and, then, gets backchased in one or more equivalent minimal queries expressed in terms of the underlying relational database.

Example 5.12: Figure 5-6 shows the integrity constraints forming Δ_{Rel} for the relational schema issued in section 5.1. There are seven constraints; the first four are primary keys and the rest are foreign keys. The maximal rewriting considered in the previous example chases with the second foreign key to

ans(y) :- Artifacts(y, x, Year, Exhibited), Painters(x, Kat), ***Paintings(y, Type)****, Artifacts(y, Painter, Year$_2$, Exhibited$_2$)*
∪ ans(y) :- Artifacts(y, x, Year, Exhibited), ***Paintings(y, Type)****, Artifacts(y, Painter, Year$_2$, Exhibited$_2$)*

Then, chases with the third foreign key to

ans(y) :- ***Artifacts(y, x, Year, Exhibited)****, Painters(x, Kat), Paintings(y, Type),* ***Artifacts(y, Painter, Year$_2$, Exhibited$_2$)****, Artists(x, Age$_1$), Artists(Painter, Age$_2$)*
∪ ans(y) :- ***Artifacts(y, x, Year, Exhibited)****, Paintings(y, Type),* ***Artifacts(y, Painter, Year$_2$, Exhibited$_2$)****, Artists(x, Age$_1$), Artists(Painter, Age$_2$)*

The first conjunctive query chases with the first foreign key constraint resulting in

ans(y) :- Artifacts(y, x, Year, Exhibited), ***Painters(x, Kat)****, Paintings(y, Type), Artifacts(y, Painter, Year$_2$, Exhibited$_2$), Artists(x, Age$_1$), Artists(Painter, Age$_2$), Artists(x, Age$_3$)*
∪ ans(y) :- Artifacts(y, x, Year, Exhibited), Paintings(y, Type), Artifacts(y, Painter, Year$_2$, Exhibited$_2$), Artists(x, Age$_1$), Artists(Painter, Age$_2$)

Applying the third primary key constraint results in

ans(y) :- Artifacts(y, x, Year, Exhibited), Painters(x, Kat), Paintings(y, Type), Artifacts(y, Painter, Year$_2$, Exhibited$_2$), Artists(x, Age$_1$), Artists(Painter, Age$_2$), Artists(x, Age$_3$), Painter=x, Year=Year$_2$, Exhibited=Exhibited$_2$
∪ ans(y) :- Artifacts(y, x, Year, Exhibited), Paintings(y, Type), Artifacts(y, Painter, Year$_2$, Exhibited$_2$), Artists(x, Age$_1$), Artists(Painter, Age$_2$), Painter=x, Year=Year$_2$, Exhibited=Exhibited$_2$

and chasing with the first primary key constraint provides the universal plan

ans(y) :- Artifacts(y, x, Year, Exhibited), Painters(x, Kat), Paintings(y, Type), Artifacts(y, Painter, Year$_2$, Exhibited$_2$), ***Artists(x, Age$_1$), Artists(Painter, Age$_2$), Artists(x, Age$_3$)****, Painter=x, Year=Year$_2$, Exhibited=Exhibited$_2$, Age1=Age2, Age2=Age3*

$\cup$ *ans(y) :- Artifacts(y, x, Year, Exhibited), Paintings(y, Type), Artifacts(y, Painter, Year*$_2$*, Exhibited*$_2$*),* ***Artists(x, Age***$_1$***)****,* ***Artists(Painter, Age***$_2$***)****,* ***Painter=x****, Year=Year*$_2$*, Exhibited=Exhibited*$_2$*, Age1=Age2*

Amazingly, this query minimises to

ans(y) :- Paintings(y, Type)

It is obvious that there can be no further minimisation of the query! ■

For this phase decidability is not guaranteed; it depends on the relational constraints. If they satisfy the stratified-witness property, the chase with Δ_{Rel} does terminate and the minimisation problem is decidable. Notice that any pair of constraints used to capture a view violates the stratified-witness condition. However, the chase is guaranteed to terminate nevertheless using the additional observation that when one of the two constraints is used in a chase step, the second one cannot trigger due to the definition of the chase step. This way the ∃-cycle in the chase flow graph breaks. Given its decidability, this phase's complexity is the one of full minimisation under DEDs.

5.2.4 Forth Phase: Translating the Query into SQL

One of the advantages of this approach is that the minimal queries resulting from the previous phases are in first-order form. Thus, they may translate into SQL queries in a straightforward manner: *(a)* the relational predicates of the query's body form the FROM clause, *(b)* the equalities involving variables introduced in any relational predicate form the WHERE clause of the SQL query, and *(c)* the head variables, except those not bound to predicates, become the projected variables in the SELECT clause with the names of the relational attributes to which they correspond. Head variables not bound to predicates get substituted by their value (found through the equalities) in the SELECT clause. This is always the case of *(i)* schema variables assigned to a value in the initial RQL querry, *(ii)* all remaining schema variables appearing in RQL patterns that have been assigned to a value during first phase, and *(iii)* variables not bound to patterns in the initial RQL$_{UCQ}$ query. The output of this translation phase is a set of SQL queries ready to be executed at remote sources.

Example 5.13: The minimal query

ans(y) :- Paintings(y, Type)

corresponds to the SQL query

```
SELECT      p.PName
FROM        Paintings p
```

based on the knowledge that *PName* is the attribute corresponding to variable *y*. Thus, this SQL query is an optimised reformulation of the RQL query

```
SELECT      Y
FROM        {X; Painter}creates{Y; Painting} ■
```

Example 5.14: In order to clarify the translation of queries carrying schema information suppose the RQL$_{UCQ}$ query

```
SELECT      $C, X
FROM        $C{X}
WHERE       $C=Artist
```

This query at the end of the first phase has the form

ans(d, x) :- C_EXT(d, x), d="Artist"

∪   ans(c, x) :- C_EXT(d, x), d="Painter", c="Artist"

which at the end of the fourth phase has become to

ans(c, x) :- Artists(x, Age), c="Artist"

The corresponding SQL query is

```
SELECT      'Artist', a.Name
FROM        Artists a
```

5.2.4.1 Handling of Functions

We have argued for the fact that the use of the *Concat* function is necessary for creating valid RDF data. Moreover, using additional functions is, also, allowed. Thus, when translating the minimal queries into SQL ones, the functional predicates have to be translated as well. However, there is no default translation for them. It depends on the SQL engine that will be used for answering the SQL query.

Example 5.15: The symbol '||' is used as the concatenation operator in Oracle10. As an alternative, Oracle10 supports a function named "*Concat*". In reality the minimal query of Example 5.12 would look something like

ans(y) :- Concat(y, "http://www.csd.uoc.gr/.../Painting.rdf#", z), Paintings(z, Type)

So, it would translate to

```
SELECT      Concat('http://www.csd.uoc.gr/.../Painting.rdf#',      p.PName)      as
PaintingURI
FROM        Paintings p
```

where "PaintingURI" is the name of the new attribute produced. ■

5.2.4.2 Choosing the Minimal Query to Be Executed

The third phase that explores relational integrity constraints may output more than one minimal query against the relational database schema. These queries shall be equivalent, i.e., if executed, they will return the same results. Thus, as soon as the translation ends, a decision has to be made concerning which of the minimal reformulated queries will be executed. This decision is not trivial and is out of the scope of this thesis. There exist various techniques that support taking such a decision based on cost models, heuristics, etc.

Generally, the existence of more than one minimal SQL queries implies redundant storage in the underlying relational database and is, usually, related to the existence of materialised views.

5.2.5 Final Phase: Translating the Results into RDF Data

Translating the results of the reformulated SQL query into RDF/S data presupposes the existence of a wrapper that will collect the results from the relational database and make the appropriate processing. The correspondence is simple. The result of executing an SQL query is a relation whose attributes are the ones projected in the query. The relation gets mapped to an rdf:Bag; if the relation has more than one attributes, each tuple corresponds to an rdf:li containing an rdf:Seq; in both cases each attribute value gets mapped to an rdf:li element.

Example 5.16: Figure 5-7 shows what the RDF/S data could look like for the minimal SQL query of the previous example. ■

```
<RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <rdf:Bag>
    <rdf:li>http://www.csd.uoc.gr/…/Painting.rdf#Guernica</rdf:li>
    <rdf:li>http://www.csd.uoc.gr/…/Painting.rdf#MonaLisa</rdf:li>
  </rdf:Bag>
</RDF>
```

Figure 5-7: RDF/S data answering the RQL$_{UCQ}$ query

5.3 Reformulation's Soundness, Completeness and Complexity

Given that the necessary conditions are met, i.e. the stratified-witness property of the relational integrity constraints, the reformulation procedure terminates and it is sound since every single step is sound, too. Moreover, all phases based on the chase and backchase algorithms are complete, too. However, the entire reformulation procedure is not complete due to the possible lack of several RDB→RDF mappings that may lead in simplifying the query to be reformulated. Finally, the reformulation's overall complexity stems from the different phases' complexities.

Chapter 6

SWIM's Architecture

As has been stressed out throughout this thesis, SWIM is a middleware for minimising and reformulating RQL queries. Figure 6-1 sketches out its general architecture, i.e. the RDF/S virtual schema and the mappings between it and the local sources (relational databases in this thesis' case). For each RQL query submitted by the user to SWIM, the reformulation engine generates several SQL queries. One of them is chosen and gets executed. Then, its results get translated using a wrapper into RDF/S data and get outputted to the user.

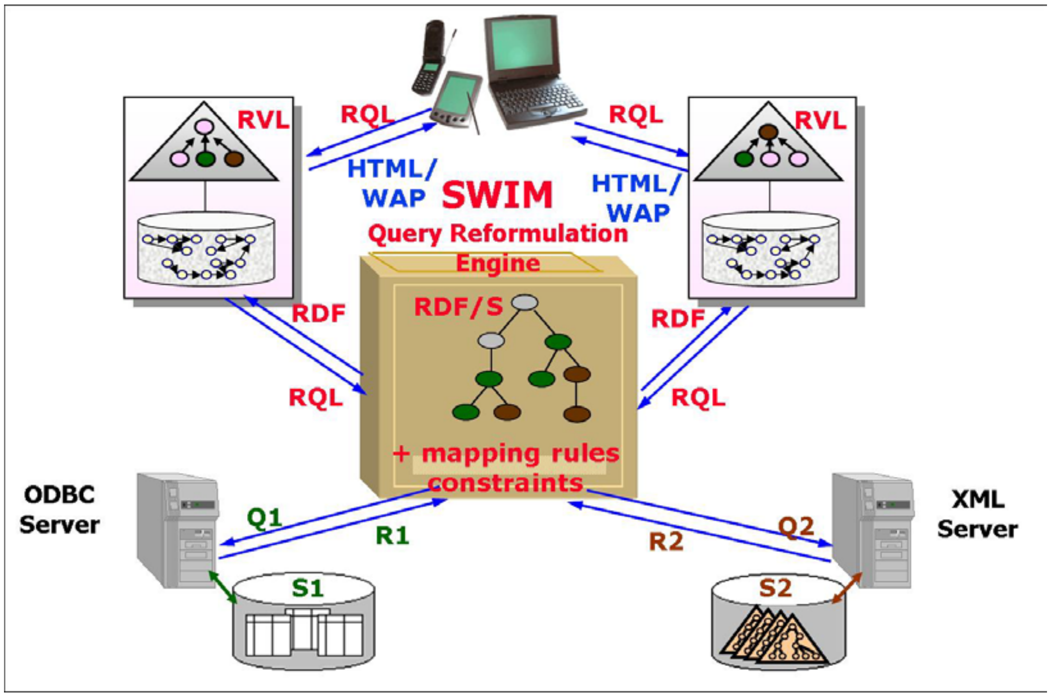


Figure 6-1: The ICS-FORTH SWIM architecture

The query reformulation engine is SWIM's core component and gets discussed below.

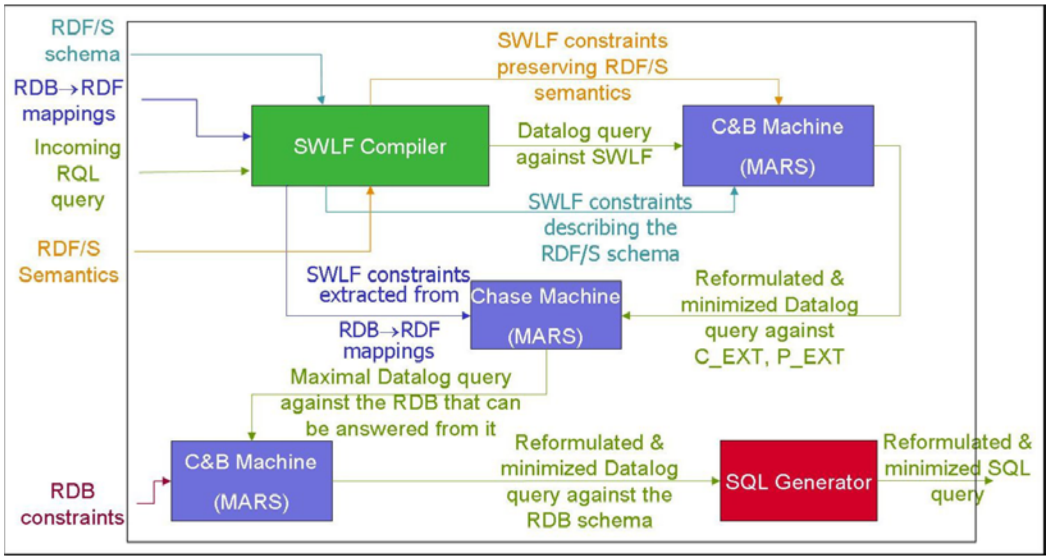

RDF/S schema
RDB→RDF mappings
Incoming RQL query
RDF/S Semantics
RDB constraints
SWLF constraints preserving RDF/S semantics
SWLF Compiler
Datalog query against SWLF
C&B Machine (MARS)
SWLF constraints describing the RDF/S schema
SWLF constraints extracted from RDB→RDF mappings
Chase Machine (MARS)
Reformulated & minimized Datalog query against C_EXT, P_EXT
Maximal Datalog query against the RDB that can be answered from it
C&B Machine (MARS)
Reformulated & minimized Datalog query against the RDB schema
SQL Generator
Reformulated & minimized SQL query

Figure 6-2: The SWIM Query Reformulation Engine

6.1 SWIM Query Reformulation Engine

The anatomy of the query reformulation engine is presented in Figure 6-2. The engine takes as input *(a)* the virtual RDF/S schema, *(b)* the RDF/S semantics, *(c)* the RDB→RDF mappings, *(d)* any constraints originating from the relational databases and *(e)* the RQL query to be reformulated.

Three basic components constitute the query reformulation engine as depicted in Figure 6-2: the SWLF compiler, MARS' engine and the SQL query generator.

6.1.1 SWLF Compiler

The SWLF Compiler takes as input all the information related to RDF and translates it in terms of SWLF. More precisely:

- It transforms the mediated RDF/S schema into Datalog facts that capture the classes, properties and their subsumption hierarchies. Then, it extracts constraints from them.
- It transforms the RQL query into a Datalog rule
- It transforms the mapping rules into rules employing SWLF terms and, then, into DEDs.

The translation of RDF/S semantics into constraints was done once and needs not be repeated each time the reformulation takes place.

Once all information is expressed as Datalog facts and rules in terms of SWLF, the Datalog program gets parsed with the use of JFlex 1.4 and JavaCup. The parsing procedure outputs a file where all information is given in the form of constraints (DEDs). The BNF grammar used for the parsing can be found in Appendix C.

6.1.2 MARS

The MARS (Mixed and Redundant Storage) system ([DT02] [DT03a] [DT03b]) constitutes the basic component of SWIM. It was developed in the University of Pennsylvania by Lucian Popa [Pop00] and Alin Deutsch [Deu02]. It implements the chase, along with the checks for consistency of the queries, and backchase algorithms referenced throughout this thesis, thus, allows checking for query equivalence and minimising as well as reformulating queries.

MARS was originally designed for object relational databases. Later, it was extended in order to handle XML repositories. This was achieved by establishing and incorporating in it a relational schema and a number of relational constraints capturing XML and its semantics. Our relational representation of RDF is somewhat similar to XML's. In our RDF to RDB scenario we ignore the XML handling feature of MARS and concentrate on the reformulation and minimisation of relational queries over the relational schema for RDF. When handling relational (and not XML) queries, the chase and backchase algorithms are sound and complete; MARS discovers all minimal (reformulated) queries.

MARS' novelty is its ability to handle both materialised views and integrity constraints. No earlier approach on reformulation had managed to prove completeness for both aspects together. What made this possible is the fact that all information is passed as input to the system in the form of constraints. Translating the mappings between the local (source) schemas and global (published) schemas into constraints allows handling both the *Global-As-View* (GAV) and the *Local-As-View* (LAV) approaches.

6.1.3 SQL Generator

The SQL Generator is the component that generates the SQL query based on the first-order query outputted from the last call of the chase/backchase machine. The translation is pretty straightforward: reading one-by-one the query's atoms, the non-functional ones are used to construct the FROM clause, the equality ones form the WHERE clause and the functional ones get incorporated in the SELECT clause. A Java program is responsible for this translation.

This translation is not independent of the underlying relational database management system (RDBMS). Since each RDBMS supports different functions or different representations of the same functions, the translation is RDBMS oriented. Each time we want to use SWIM over a RDBMS for the first time, the generator has to be enriched with the appropriate methods handling the RDBMS' functions.

6.2 Related Work

RDF/S is a SW language that has become accepted as a language favouring interoperability between information sources in the last few years. However, the number of systems having adopted RDF/S in order to integrate/publish relational sources in the Semantic Web is still limited. Nevertheless, many systems are under development and new approaches are proposed daily. Some of them republish entire relational databases as RDF/S data adopting the data warehousing approach, while the rest, like SWIM, return virtual RDF resource descriptions when queries are posed on virtual RDF/S schemas (on-demand retrieval).

The situation is even worse with approaches proposing and systems offering containment and minimisation techniques for RDF/S query languages. To our knowledge there exists only one application offering such services and there is no system providing optimisation services along with the integration ones. Moreover, there is a second one that deals with the idea of query caching, i.e. reusability of previously computed results.

6.2.1 SWARD

SWARD [PR04] is a system under development that offers wrapping services to relational databases by adopting the on-demand retrieval approach. This is achieved by extracting virtual RDF/S views from the relational databases; these views, which are defined using domain calculus expressions, express relational to RDF mappings. RDF/S queries – QEL ([NS04]) is favoured as the query language for RDF/S – get translated to domain calculus expressions, too, and get composed with the RDF/S views. Then, the resulting domain calculus expression gets translated to an SQL query whose results are returned as virtual RDF resource descriptions. SWARD allows using in the RDF/S queries filters not expressible in SQL; in this case, the filters are applied on the results from the SQL queries.

Moreover, SWARD allows for different terminologies in received QEL queries by maintaining a user defined table that stores relations between terms from different ontologies having the same meaning; this way the RDB-specific RDF/S views can map to global ontologies. When a query uses terms of the global ontology, the system searches the table for equivalent terms and rewrites the query.

SWARD's basic drawback is the limited expressive power of the mappings relating a relational database to an RDF/S ontology. The RDF/S views follow the GAV approach and are provided only in terms of virtual RDF/S properties, i.e. of basic RDF/S data; no schema information about instance-of, class-property hierarchies can be expressed. Moreover, the correspondences between RDF/S views and global ontologies are elementary, since they are based on term equivalence.

6.2.2 D2RQ

D2RQ ([BS04]) is a declarative, based on D2R Map ([Bi03]), language used to describe mappings between relational database schemata and RDF/S (OWL) ontologies. It is used as an add-in to the Jena tookit ([CDD$^+$03]) rendering on-demand retrieval feasible. It allows treating the relational databases as virtual RDF graphs, which can be queried using RDQL; the queries based on the mappings get reformulated to SQL queries, whose results get translated back to RDF data.

A D2RQ mapping between a global ontology and a relational database schema is an RDF/S document that describes *(a)* correspondences of RDF/S classes and properties to relational elements; based on these correspondences class and property instances get extracted on demand, and *(b)* how class instances are identified, i.e. how URIs for instances are produced. This work is pretty new and under development.

D2RQ follows the GAV approach in the definition of the mappings. Therefore, compared to our work, the mappings are less expressive.

6.2.3 Integration of Relational Sources Using RDF and XML

[HV01] proposes an architecture for integrating heterogeneous information sources using RDF and XML. This on-demand retrieval approach is based on a conceptual domain model described using RDF/S. Mappings are established between the virtual RDF/S schema and virtual or not XML DTDs using mapping rules expressed in LMX[18]. Therefore, information sources are required to be able to export their data in XML serialisation and non-XML sources, like relational databases, can use wrappers to achieve that. Once a query is posed on the conceptual model it gets pushed to the underlying sources based on the mappings; then, the sources, with or without a wrapper's interference, return all necessary information as XML serialisation and RDF/S data get produced.

6.2.4 Integration of Relational Sources using RDF Vocabularies

[CX03] introduces an approach involving both query reformulation and data warehousing. Queries are posed on a local source described by an RDF/S schema and get mapped to the remote source, described by an RDF/S schema, too; RQL has been adopted as the query language. The mapping is facilitated by a global RDF/S ontology and common vocabularies. Each source schema shares with the global ontology a dictionary that stores the common vocabulary of all the schema concepts and the relationships between the ontology and each schema.

The interesting part is that the sources originally need not have an RDF/S schema describing them and their data stored as RDF. If they are relational (or XML), an RDF/S schema gets extracted and RDF/S data are created and stored into RSSDB

[18] Language for Mapping XML documents

([ACK$^+$01]). Thus, as soon as an RQL query on the local source gets reformulated to an RQL query on the target source it can get executed.

The main difference with SWIM is the fact that the integration of relational sources into global or other RDF/S ontologies demands that an RDF/S schema along with RDF/S data gets extracted. Moreover, a basic restriction of this approach that is worth noticing is the fact that only one-to-one mappings are considered when relating each source's RDF/S schema to the global ontology, i.e., a concept in one RDF/S schema maps to a single concept in another schema.

6.2.5 FDR2

In [KT04] the authors propose a data warehousing approach to facilitate ontology-based querying of relational data. The goal is to relate data stored in a relational database with a domain ontology. The first step is to extract an RDF/S representation of the relational database. Automatically, every column of a relational table gets mapped to an RDF/S class and all binary relations between two columns (classes) get mapped to RDF/S properties. Then, all classes and properties get instantiated using the relational data. The next step is to map this RDF/S representation to the domain ontology, which is also expressed in RDF/S. The user identifies subclass (subproperty) relationships between the classes (properties) of the RDF/S representation and those of the domain ontology. As soon as the mapping is done, an RDF/S reasoner is used to deduct all possible entailments based on the hierarchy relationships. Whenever a query on the domain ontology is posed, it gets evaluated using these entailments.

FDR2 uses a naïve approach for creating the RDF/S representation, since it does not explore its semantics. This choice was made because it is designated for simple relational schemas, like the ones used for keeping track of scientific experiments and computations. Moreover, based on the mappings, we may conclude that, using the FDR2 approach, queries built on proper interpretations cannot be answered. Finally, FDR2 provides no query optimisation techniques.

6.2.6 D2R Map

D2R Map ([Bi03]) is a declarative XML-based language used to describe flexible mappings of complex relational structures to RDF. Its flexibilty is achieved by employing SQL statements directly in the mapping rules; correspondences between classes (properties) and relational elements are established stating how the classes (properties) get instantiated and how URIs are created. This way, relational data can get exported as RDF using a D2R processor.

6.2.7 ICS-FORTH GRQL Interface

As has already been illustrated, the ICS-FORTH GRQL Interface ([ACK04]) is an interface that produces on the fly RQL queries while the user navigates through an RDF/S schema. Its functionality is rather simple: each navigation step on an RDF/S schema either creates or alters an existing path expression; these path expressions are combined to form RQL queries. One of its advantages is that it performs optimisation of the queries produced run-time. The key to optimisation is that when navigating through hierarchies of classes (properties) already present in path expressions, the path expressions get refined depending on whether subclasses (subproperties) are visited. The main difference to SWIM's query minimisation lays to its handling of only a fragment of RQL, similar to RQL_{CORE}.

6.2.8 Similarity-Based Query Caching

In [Stu04] the author approaches RDF query optimisation from a different perspective. He proposes a graph-based approach for identifying RDF queries that are subsumed by already issued queries whose results have been cached (stored). This approach exploits the fact that RDF statements form labelled directed graphs. Queries are represented using graphs where the unlabelled nodes denote the variables of the query. By comparing query graphs query subsumption can be identified. If query *A*, which was issued on the RDF/S description base *DB* subsumes query *B*, query *B* needs not be executed on *DB*. It can be issued on the cached results of query *A*.

Additionally, the author provides *(i)* a cost model to decide whether result caching provides an advantage with respect to run time complexity and *(ii)* a cost-based similarity measure for RDF queries in case more than one relevant result sets are found in the cache.

The key difference to our minimisation approach is that the graph queries can be posed only on nodes, i.e. demand subject/object resource information. In other words only queries asking solely for data information though joining property interpretations are supported.

Chapter 7

Conclusion

The issue of integrating legacy systems has been challenging information society for a long time. Initially, systems were integrated by defining one-to-one mappings between them. However, this approach presented several drawbacks, like scalability and maintainability. Then, organisations moved to XML in order to take advantage of standards based integration. But, XML did not provide solutions to all problems.

The next step was to use ontologies in the integration process. For each knowledge or application domain, an ontology is defined and legacy systems get mapped to such ontologies. This way, systems can communicate independently from information architectures and system technologies. Moreover, when ontologies are described using machine processable languages, like RDF/S, legacy systems get integrated into the Semantic Web and can be accessed through it.

Therefore, in this thesis we have proposed SWIM, a system for integrating relational and XML ([Kof05]) sources in the Semantic Web, and have focused on the relational aspects of the system. At the same time we have presented the optimisation capabilities of SWIM regarding queries against SW ontologies. We have chosen RDF/S to be the SW ontology language and RQL its corresponding query language. We have, also, identified a fragment of RQL, namely RQL_{CORE}, for which optimisation is sometimes easier to perform.

Six relational predicates and a number of constraints, which form the Semantic Web Logic Framework (SWLF), have been adopted for capturing the RDF/S data model and its semantics. Based on this representation we have achieved to reduce the RQL optimisation problem to its relational equivalent. Furthermore, the RQL to SQL reformulation problem has been reduced to the problem of rewriting a query against SWLF as a query against the relational storage schema.

The minimisation and reformulation procedures are based on the chase and backchase algorithms; the latter is sound and complete when the chase is guaranteed to terminate. Since these algorithms accept input in the form of constraints, all information gets translated into disjunctive embedded dependencies (DEDs). The reformulation procedure exploits RDB to RDF mappings translated in SWLF. The mappings between the local (relational) sources and the global (RDF) ones follow the GLAV approach that combines the advantages of both the Global-As-View and Local-As-View approaches.

It is worth noticing that the chase and backchase algorithms employed for RDF/S semantic query optimisation were initially developped in the context of optimisation and reformulation of queries issued against relational schemas using dependencies capturing integrity constraints and schema mappings. However, in our scenario the major difference that arises is the expressions of queries allowing both schema and data navigation/filtering. In order to represent a specific RDF/S description schema, the predicates CLASS, PROP, C_SUB and P_SUB have to get populated. Moreover, this schema is given as input to the chase and backchase algorithms using constraints, which implies providing the data of the relations in the form of constraints. Using this knowledge, the minimisation process always generates a minimal query were no schema navigation is needed in order to answer it. In practice, the minimisation procedure sometimes rather answers than minimises the schema part of a given query; when the case, a union is usually introduced.

Additionally, not all the minimal queries produced from the C&B algorithm are always interesting. For example, it is hard to argue why choosing to execute the second minimal query of Example 4.20. If accessing only proper interpretations is desired (i.e. the relation C_EXT), the third query should be chosen; if accessing extended interpretations is desired (i.e. both the C_SUB and C_EXT relations), the first query should be chosen. The only obvious reason to select a minimal query where schema information has partially been unfolded is in order to exploit cached query results; if the query asking for the extended interpretation of *Painter* – or the proper interpretation of either *Artist* or *Sculptor* – has already been issued and its results have been stored, this query can be useful. Nevertheless, this presupposes a check of all the minimal queries in order to locate those involving cached queries.

Another issue has occurred in the reformulation process. If we were interested in considering only GAV mappings, the reformulation would be straightforward even if they were not complete, i.e. did not provide information for all classes and properties of the RDF/S schema. However, our goal was to consider GLAV mappings between the virtual RDF/S schema and the relational proprietary one; so, we were forced to break the reformulation in three phases in order to handle/overcome mappings' incompleteness.

This incompleteness usually does not affect the straightforwardness of the reformulation algorithm in purely relational scenarios. When a query does not reformulate as a whole, a negative answer is issued. On the contrary, in the RQL reformulation/integration scenario the majority of RQL conjunctive queries imply a union of queries based on the class/property hierarchies. The underlying proprietary database may not be able to answer all of them, but may have the information to answer some of them. Our reformulation algorithm manages to handle this case, too, by locating the maximal subquery of the original one that can be answered.

7.1 Future Work

SWIM is a middleware for RQL query optimisation and reformulation. However, there are still some issues that deserve to be further investigated.

First of all, it would be very useful if SWIM disposed a feature that would allow generating (semi-) automatically the mappings between the virtual RDF/S schemata and the relational databases; users would appreciate such assistance. A possible guideline to this direction could be to incorporate in SWIM a reverse engineering tool in order to produce an ER-model from the relational schema; given the concepts and the relationships between them, the mapping to RDF/S classes and properties would be facilitated.

Another issue is the support of features originating from more expressive RDF-based ontology languages, like OWL. OWL offers some additional features that can prove very useful, such as inverse properties and disjointness of class and property extensions. Incorporating them in SWIM would allow a wider range of RQL queries to be posed and would offer increased optimisation capabilities.

Additionally, extending the RQL fragment considered, namely RQL_{UCQ} could be a hint for future work. For example, we could look into incorporating RQL functions – like domain, range, subclassof, subpropertyof and aggregate ones – and nested queries in the fragment.

The ability to attach to the reformulated queries schema information is an issue that deserves further investigation, too; although the queries are posed against an

RDF/S schema, the algorithm’s present form does not allow, when asking a query, retrieving along with the instances the corresponding schema information, i.e. returning fully typed descriptions. Such a feature would allow exploiting the power of the RDF/S language.

Another interesting direction for further investigation is the exploitation of cached query results either in combination with the minimisation process or independently. The problem can be defined as follows: “Given a query or a minimal equivalent, can it partially (or fully) be answered from the cached results of already answered queries?”

Finally, extended paradigms should be run in order to test whether SWIM can be used in real life integration scenarios where the input load may severally increase. We should have in mind that the complexities raise exponentially in the size of the global RDF/S schema.

Bibliography

[ACK$^+$01] Sofia Alexaki, Vassilis Christophides, Gregory Karvounarakis, Dimitris Plexousakis, and Karsten Tolle. *The ICS-FORTH RDFSuite: Managing Voluminous RDF Description Bases*. In Proceedings of the 2nd International Workshop on the Semantic Web, Hong Kong, 2001.

[ACK04] Nikos Athanasis, Vassilis Christophides, and Dimitris Kotzinos. *Generating on the Fly Queries for the Semantic Web: The ICS-FORTH Graphical RQL Interface (GRQL)*. In Proceedings of the 3rd International Semantic Web Conference, pages 486-501, Hiroshima, Japan, 2004.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[Bec04] Dave Beckett. *RDF/XML Syntax Specification*. W3C Recommendation, 10 February 2004.

[BG04] Dan Brickley, and R.V. Guha. *RDF Vocabulary Description Language 1.0: RDF Schema*. W3C Recommendation, 10 February 2004.

[Bi03] Christian Bizer. *D2R Map – A Database to RDF Mapping Language*. Poster at the 12th World Wide Web Conference, Budapest, Hungary, 2003.

[BMPS00] Tim Bray, Eve Maler, Jean Paoli, and C. M. Sperberg-McQueen. *Extensible Markup Language (XML) 1.0*. W3C Recommendation, 6 October 2000.

[BS04] Christian Bizer, and Andy Seaborne. *D2RQ – Treating Non-RDF Databases as Virtual RDF Graphs*. 3rd International Semantic Web Conference, Hirosima, Japan, 2004.

[CDD+03] Jeremy J. Carroll, Ian Dickinson, Chris Dollin, Dave Reynolds, Andy Seaborne, and Kevin Wilkinson. *Jena: Implementing the Semantic Web Recommendations*. Technical Report, HP Labs, 2003.

[CK04] Jeremy J. Carroll, and Graham Kline. *Resource Description Framework (RDF): Concepts and Abstract Syntax*. W3C Recommendation, 10 February 2004.

[CM77] Ashok K. Chandra, and Philip M. Merlin. *Optimal Implementation of Conjunctive Queries in Relational Data Bases*. In Proceedings of the 9th Annual ACM Symposium on the Theory of Computing, pages 77-90, 1977.

[CX03] Isabel F. Cruz, and Huiyong Xiao. *Using a Layered Approach for Interoperability on the Semantic Web*. In Proceedings of the 4th International Conference on Web Information Systems Engineering (WISE), pages 221-231, 2003.

[Deu02] Alin Deutsch. *XML Query Reformulation over Mixed and Redundant Storage*. PhD Thesis, University of Pennsylvania, 2002.

[DT02] Alin Deutsch, and Val Tannen. *Querying XML with Mixed and Redundant Storage*. Technical Report, University of Pennsylvania, 2002.

[DT03a] Alin Deutsch, and Val Tannen. *MARS: A System for Publishing XML from Mixed and Redundant Storage*. In Proceedings of the 29th VLDB Conference, Berlin, Germany, 2003.

[DT03b] Alin Deutsch, and Val Tannen. *Reformulation of XML Queries and Constraints*. In Proceedings of the 9th International Conference on Database Theory (ICDT), Siena, Italy, 2003.

[FLM99] Mark Friedman, Alon Y. Levy, and Todd Milstein. *Navigational Plans for Data Integration*. In Proceedings of the 16th National Conference on Artificial Intelligence and 11th Conference on Innovative Applications of Artificial Intelligence, pages 67-73, 1999.

[GGGM98] Parke Godfrey, John Grant, Jarek Gryz, and Jack Minker. *Integrity Constraints: Semantics and Applications*. In Logics for Databases and Information Systems, pages 265-306, 1998.

[Hay04] Patrick Hayes. *RDF Semantics*. W3C Recommendation, 10 February 2004.

[HM04] Frank Van Harmelen, and Deborah L. McGuinness. *OWL Web Ontology Language Overview*. W3C Recommendation, 10 February 2004.

[HV01] Geert-Jan Houben, and Richard Vdovjak. *RDF Based Architecture for Semantic Integration of Heterogeneous Information Sources*. International Workshop on Information Integration on the Web (WIIW), Rio de Janeiro, Brazil, 2001.

[Kar00] Grigoris Karvounarakis. *Querying RDF Metadata for Community Web Portals*. Master Thesis, University of Crete, 2000.

[Kof05] Ioanna Koffina. *Integrating XML Data Sources Using RDF/S Schemas: The ICS-FORTH Semantic Web Integration Middleware (SWIM)*. Master Thesis, University of Crete, 2005.

[KT04] Maksym Korotkiy, and Jan L. Top. *From Relational Data to RDFS Model*. International Conference on Web Engineering, Munich, Germany, 2004.

[Lev99] Alon Y. Levy. *Logic-Based Techniques in Data Integration*. Workshop on Logic-Based Artificial Intelligence, Washington, USA, 1999.

[Lev01] Alon Y. Levy. *Answering Queries Using Views: A Survey*. The International Journal on Very Large Databases, 2001.

[Mag03] Aimilia Magkanaraki. *A View Definition Language for RDF/S*. Master Thesis, University of Crete, 2003.

[MM04] Frank Manola, and Eric Miller. *RDF Primer*. W3C Recommendation, 10 February 2004.

[NS04] Mikael Nilsson, and Wolf Siberski. *RDF Query Exchange Language (QEL) – Concepts, Semantics and RDF Syntax*. http://edutella.jxta.org/spec/qel.html.

[Ono05] Nicola Onose. *Extensions of the Relational Chase*. Project Report of End of Studies, 2005.

[Pop00] Lucian Popa. *Object/Relational Query Optimisation with Chase and Backchase*. PhD Thesis, University of Pennsylvania, 2000.

[PR04] Johan Petrini, and Tore Risch. *Processing Queries over RDF Views of Wrapped Relational Databases*. In Proceedings of the 1st International Workshop on Wrapper Techniques for Legacy Systems (WRAP), Delft, Netherlands, 2004.

[PS05] Eric Prud'hommeaux, and Andy Seaborne. *SPARQL Query Language for RDF*. W3C Working Draft, 17 February 2005.

[Stu04] Heiner Stuckenschmidt. *Similarity-Based Query Caching*. In Proceedings of the 6th International Conference on Flexible Query Answering Systems (), Lyon, France, 2004.

[SW01] Tim Berners-Lee, James Hendler, and Ora Lassila. *The Semantic Web: A new form of Web content that is meaningful to computers will unleash a revolution of new possibilities*. Scientific American, 17 May 2001. Available at http://www.scientificamerican.com/print_version.cfm?articleID=00048144-10D2-1C70-84A9809EC588EF21

[Ull00] Jeffrey D. Ullman. *Information Integration Using Logical Views*. Theoretical Computer Science, 239(2): pages 189-210, 2000.

Appendix A

RQL Patterns

RQL Property Patterns

The table containing all property patterns appearing in RQL_{UCQ} queries and their translations in SWLF follows.

Property Pattern	**Translation**
@P *^@P*	*PROP(a, p, b)*
{X; ^$C}@P{Y; ^$D}	*PROP(a, p, b), P_SUB(q, p), P_EXT(x, q, y), C_SUB(c, a), C_SUB(d, b), C_EXT(c, x), C_EXT(d, y)*
{X; ^$C}@P{Y}* *{X; ^$C}@P	*PROP(a, p, b), P_SUB(q, p), P_EXT(x, q, y), C_SUB(c, a), C_EXT(c, x)*

{X}@P{Y; ^$D}
@P{Y; ^$D}
	PROP(a, p, b), P_SUB(q, p), P_EXT(x, q, y), C_SUB(d, b), C_EXT(d, y)

{X; $C}@P{Y; ^$D}
{$C}@P{Y; ^$D}
	PROP(a, p, b), P_SUB(q, p), P_EXT(x, q, y), C_SUB(c, a), C_SUB(d, b), C_SUB(e, c), C_EXT(e, x), C_EXT(d, y)

{X; ^$C}@P{Y; $D}
{X; ^$C}@P{$D}
	PROP(a, p, b), P_SUB(q, p), P_EXT(x, q, y), C_SUB(c, a), C_SUB(d, b), C_SUB(f, d), C_EXT(c, x), C_EXT(f, y)

{X; $C}@P{Y; $D}
{$C}@P{Y; $D}
{X; $C}@P{$D}
	PROP(a, p, b), P_SUB(q, p), P_EXT(x, q, y), C_SUB(c, a), C_SUB(d, b), C_SUB(e, c), C_SUB(f, d), C_EXT(e, x), C_EXT(f, y)

{X; $C}@P{Y}
{$C}@P{Y}
{X; $C}@P
	PROP(a, p, b), P_SUB(q, p), P_EXT(x, q, y), C_SUB(c, a), C_SUB(e, c), C_EXT(e, x)

{X}@P{Y; $D}
{X}@P{$D}
@P{Y; $D}
	PROP(a, p, b), P_SUB(q, p), P_EXT(x, q, y), C_SUB(d, b), C_SUB(f, d), C_EXT(f, y)

{X}@P{Y}	*P_SUB(q, p), P_EXT(x, q, y)*

{X}@P *@P{Y}*	

{$C}@P{$D}
{$C}^@P{$D}
	PROP(a, p, b), C_SUB(c, a), C_SUB(d, b),

*{$C}@P* {$C}^@P	*PROP(a, p, b), C_SUB(c, a)*

@P{$D}
^@P{$D}
	PROP(a, p, b), C_SUB(d, b)

{X; ^$C}^@P{Y; ^$D}	PROP(a, p, b), P_EXT(x, p, y), C_SUB(c, a), C_SUB(d, b), C_EXT(c, x), C_EXT(d, y)
{X; ^$C}^@P{Y} {X; ^$C}^@P	PROP(a, p, b), P_EXT(x, p, y), C_SUB(c, a), C_EXT(c, x)

{X}^@P{Y; ^$D}
^@P{Y; ^$D}
	PROP(a, p, b), P_EXT(x, p, y),
	C_SUB(d, b), C_EXT(d, y)

{X; $C}^@P{Y; ^$D}
{$C}^@P{Y; ^$D}
	PROP(a, p, b), P_EXT(x, p, y),
	C_SUB(c, a), C_SUB(d, b), C_SUB(e, c), C_EXT(e, x),
	C_EXT(d, y)

{X; ^$C}^@P{Y; $D}
{X; ^$C}^@P{$D}
	PROP(a, p, b), P_EXT(x, p, y),
	C_SUB(c, a), C_SUB(d, b), C_SUB(f, d), C_EXT(c, x),
	C_EXT(f, y)

{X; $C}^@P{Y; $D}
{$C}^@P{Y; $D}
{X; $C}^@P{$D}
	PROP(a, p, b), P_EXT(x, p, y),
	C_SUB(c, a), C_SUB(d, b), C_SUB(e, c), C_SUB(f, d),
	C_EXT(e, x), C_EXT(f, y)

{X; $C}^@P{Y}
{$C}^@P{Y}
{X; $C}^@P
	PROP(a, p, b), P_EXT(x, p, y),
	C_SUB(c, a), C_SUB(e, c), C_EXT(e, x)

{X}^@P{Y; $D}
{X}^@P{$D}
^@P{Y; $D}
	PROP(a, p, b), P_EXT(x, p, y),
	C_SUB(d, b), C_SUB(f, d), C_EXT(f, y)

{X}^@P{Y}
{X}^@P
^@P{Y}
	P_EXT(x, p, y)

RQL Class Patterns Not Involving Proper Interpretations

The table containing all class patterns appearing in RQL$_{CORE}$ queries follows.

Class Pattern	
$C	*^$C*
$C{$D}	*^$C{$D}*
$C{X}	
$C{X; $D}	

RQL Property Patterns Not Involving Proper Interpretations

The table containing all class patterns appearing in RQL$_{CORE}$ queries follows.

Property Pattern		
@P	^@P	
{X; $C}@P{Y; $D}	{$C}@P{Y; $D}	{X; $C}@P{$D}
{X; $C}@P{Y}	{$C}@P{Y}	{X; $C}@P
{X}@P{Y; $D}	{X}@P{$D}	@P{Y; $D}
{X}@P{Y}	*{X}@P*	*@P{Y}*
{$C}@P{$D}	{$C}^@P{$D}	
{$C}@P	{$C}^@P	
@P{$D}	^@P{$D}	

Appendix B

Termination of Chase

Stratified-Witness for Disjunction-free DEDs

We will start by illustrating why the stratified-witness property can be used with disjunction-free DEDs, i.e. DEDs consisting of a single conjunctive query. First of all, we will consider equalities in the left-hand side of dependencies. Such equalities can easily be ignored by replacing all occurrences in the dependency of one of the variables equated by the other, hence, resulting in an embedded dependency as introduced in [AHV95].

Equalities in the right hand of a dependency involving at least one existentially quantified constraint are dealt similarly. If two existentially quantified variables are equated, the equality is removed and one of the variables is replaced by the other one in the right-hand side of the dependency. If an existentially quantified variable is equated with a universally quantified one, then, the equality is removed and the universally quantified one replaces the other one. Once again the query produced is an embedded dependency equivalent to the original dependency.

Finally, we will argue that the introduction of constants in the dependencies cannot disturb the stratified-witness property. We can without loss of expressive power consider constants only in equality atoms. If a set of dependencies, where equality atoms involving constants are ignored, satisfies the stratified-witness property and, therefore, termination is ensured, there is no way the introduction of the unattended equalities will introduce an infinite number of fresh variables and lead the chase to diverge.

Proposition B.1: *If a set C of disjunction-free DEDs satisfies the stratified-witness property when equalities of variables to constants are ignored, the chase of a conjunctive query Q with C terminates.* ■

In practice, the use of constants in the constraints may reduce the number of chase steps till the universal plan is reached, or even prevent an endless execution of chase steps. Unfortunately, the check for stratified-witness cannot exploit the additional information coming with the use of constants.

Example B.1: The constraints below do not satisfy the stratified-witness.

(d_1) $\forall x \forall y\ A(x, y) \rightarrow \exists z\ B(y, z)$
(d_2) $\forall y \forall z\ B(x, y) \rightarrow \exists z\ A(y, z)$

and the chase of query

ans(x) :- A(x, y)

with them will not terminate

ans(x) :- A(x, y), B(y, z)
→ ans(x) :- A(x, y), B(y, z), A(z, z1)
→ ans(x) :- A(x, y), B(y, z), A(z, z1), B(z1, z2)
→ …

If we refine (*d_1*) by adding the equality "z=3", the constraints do not satisfy the stratified-witness property once again, since the process to verify it has not changed. Nevertheless, the chase with the new constraints terminates.

ans(x) :- A(x, y), B(y, 3)
→ ans(x) :- A(x, y), B(y, 3), A(3, k)
→ ans(x) :- A(x, y), B(y, 3), A(3, k), B(k, 3)

The chase terminates here. ■

Termination of Chase with Δ_{Map}

Initially, we will argue that the constraints issued from a single mapping rule do not cause the chase to diverge, although they introduce at least one cycle in the chase flow graph that contains at least one ∃-labelled edge. The two constraints ensuring soundness and completeness (see section 5.1.1) create such a cycle. However, they cannot trigger both; when one of them gets applied, the conclusion of the second one is already in the query. We have, also, seen that several constraints, whose head is a subset of the constraint ensuring completeness, can be educed. Incorporating these constraints may introduce new cycles containing an ∃-labelled edge. Nevertheless, the chase will terminate because each additional constraint can apply at most once, depending on the order the constraints get applied.

Example B.2: Suppose the mapping rule

$P_1(x, y)$, $P_2(y, z)$:- R(x, y, z, …)

In SWLF it takes the form

P_EXT(x, P_1, y), P_EXT(y, P_2, z) :- R(x, y, z, …)

The constraints extracted are

(d_1) ∀x∀y∀z∀p∀q P_EXT(x, p, y) ^ P_EXT(y, q, z) ^ p="P_1" ^ q="P_2" → ∃... R(x, y, z, ...)
(d_2) ∀x∀y∀z∀… R(x, y, z, …) → ∃p∃q P_EXT(x, p, y) ^ P_EXT(y, q, z) ^ p="P_1" ^ q="P_2"
(d_{11}) ∀x∀y∀p P_EXT(x, p, y) ^ p="P_1" → ∃z∃... R(x, y, z, ...)
(d_{12}) ∀y∀z∀q P_EXT(y, q, z) ^ q="P_2" → ∃x∃... R(x, y, z, ...)

Suppose the query below

ans(x) :- P_EXT(x, p, y), P_EXT(y, q, z), p="P_1", q="P_2"

If d_1 is applied first, the chase ends with the universal plan

ans(x) :- P_EXT(x, p, y), P_EXT(y, q, z), p="P_1", q="P_2", R(x, y, z, ...)

On the contrary, if d_{11} (or d_{12}) is applied first, d_1 gets applied, too. Then, the universal plan looks like

ans(x) :- P_EXT(x, p, y), P_EXT(y, q, z), p="P_1", q="P_2", R(x, y, z', ...), R(x, y, z, ...)

Alternatively, we may apply d_{11} and d_{12}, then d_2 twice and, finally, d_1 twice.

ans(x) :- P_EXT(x, p, y), P_EXT(y, q, z), p="P_1", q="P_2", R(x, y, z', …), R(x', y, z), P_EXT(y, q, z'), P_EXT(x', p, y), R(x', y, z'), R(x, y, z)

After a number of steps and independently of the order the constraints get applied, the chase terminates. ■

The conclusion above propagates to any mapping; the constraints extracted from a single mapping cannot impose an infinite number of chase steps. Obviously, the number of the chase steps depends on the number of RVL clauses appearing in the left-hand side of the mapping, which produce an equal number of constraints. The previous example along with other ones led us to the conclusion that whenever possible, first the constraints of the form d_1, then those of the form d_2 and finally those of the form d_{1i} should be applied in order to reduce the number of the chase steps.

Now, we will extend our reasoning in order to comprise the interaction between the constraints originating from different mappings. The fact that all class (property) extents are expressed using a single predicate renders the emergence of cycles containing ∃-labelled edges very possible in the chase flow graph. However, notice that the variables corresponding to the class (property) names in the C_EXT (P_EXT) predicates are always assigned to a constant in the query and the mappings and, consequently, the constraints. Therefore, given a chase sequence of the form

$$C\_EXT(c_1, x) \xrightarrow{\delta_1} \ldots \xrightarrow{\delta_i} \ldots \xrightarrow{\delta_n} C\_EXT(c_2, y)$$

or

$$P\_EXT(x, p_1, y) \xrightarrow{\delta_1} \ldots \xrightarrow{\delta_i} \ldots \xrightarrow{\delta_n} P\_EXT(x', p_2, y')$$

and in order to have an infinite number of chase steps, the equality $c_1=c_2$ ($p_1=p_2$) must stand.

Having this in mind, we can use the following technique to test the constraints for stratified-witness: for every distinct value of the class (property) name in the extent predicates we introduce a "virtual" predicate; the chase flow graph is built using the "virtual" predicates and not the C_EXT and P_EXT ones. For example, the predicate P_EXT(x, creates, y) is handled in the chase flow graph as creates(x, y). This trick we use is not as arbitrary as it seems; in fact it is the opposite procedure from the one taking place when translating the RVL clauses into SWLF. Testing dependencies for stratified-witness using this technique overcomes the inability of exploiting the values of the class (property) names in the C_EXT (P_EXT) predicates.

Example B.3: Suppose the mappings

P_EXT(x, creates, y) :- Artifacts(y, x, Exhibited, Year)
P_EXT(x, isDated, x) :- Artifacts(x, Artist, Exhibited, y)

If, after translating them into constraints, we test the latter for stratified-witness, the result will be negative; there are cycles containing at least one ∃-labelled edge. If, however, the aforementioned trick is used, the "altered" constraints satisfy stratified-witness and we are able to recognise that the chase terminates. ■

Although the technique above allows us to overcome the fact that the check for stratified-witness cannot handle the values in the C_EXT and P_EXT predicaets, there are still cases that the check cannot identify as terminating due to not handling values assigned to variables. Nevertheless, the fact that the constraints are extracted from the mappings using a standard procedure allows a very important observation: the chase with Δ_{Map} always terminates! We will argue for this observation based partly on intuition.

Since all constraints have on one side C_EXT (P_EXT) predicates and on the other relational predicates, every infinite chase sequence, if there could exist one, should introduce infinite C_EXT (P_EXT) predicates. Moreover, the constraints creating an "∃-cycle" in the chase flow graph are of even number. We will show why there cannot be an infinite number of chase steps when two constraints create a cycle in their chase flow graph using general examples. The same proof procedure can be used when four or more constraints create such a cycle.

Example B.4: We will first consider the case where the cycle results in an infinite number of C_EXT predicates. For this to happen the constraints creating the cycle should look like

$\forall c\, \forall x\; C\_EXT(c, x) \wedge c=\text{“C”} \rightarrow \exists y\; R(x, y, \ldots)$
$\forall x\, \forall y\; R(x, y, \ldots) \rightarrow \exists c\; C\_EXT(c, y) \wedge c=\text{“C”}$

If these were not constraints extracted from mappings the chase of query

ans(x) :- C_EXT(C, x)

would diverge:

ans(x) :- C_EXT(C, x), R(x, y, …), C_EXT(C, y), R(y, y', …), C_EXT(C, y'), …

However, the above two constraints imply the existence of the following mapping rules[19]

C_EXT(C, x) :- R(x, y, …)
C_EXT(C, x) :- R(y, x, …)

The constraints that would be given as input to the chase algorithm would be

$\forall c \forall x\ C\_EXT(c, x) \wedge c=“C” \rightarrow (\exists y\ R(x, y, \ldots)) \vee (\exists y\ R(y, x, \ldots))$
$\forall x \forall y\ R(x, y, \ldots) \rightarrow \exists c\ C\_EXT(c, y) \wedge c=“C”$
$\forall x \forall y\ R(x, y, \ldots) \rightarrow \exists c\ C\_EXT(c, x) \wedge c=“C”$

These constraints violate the stratified-witness, too. All the same, the chase would terminate yielding the universal plan

ans(x) :- C_EXT(C, x), R(x, y, …), C_EXT(C, y)
$\cup$ *ans(x) :- C_EXT(C, x), R(y, x, …), C_EXT(C, y)* ■

Example B.5: Now, let’s study study the case where the cycle containing the ∃-labelled edge involves P_EXT. Two constraints that would result in an infinite number of chase steps are

$\forall x \forall p \forall y\ P\_EXT(x, p, y) \wedge p=“P_1” \rightarrow \exists z\ R(x, y, z, \ldots)$
$\forall x \forall y \forall z\ R(x, y, z, \ldots) \rightarrow \exists p\ P\_EXT(y, p, z) \wedge p=“P_1”$

Imagine the query

ans(x) :- P_EXT(x, P_1, y)

Its chase with the constraints above would diverge

ans(x) :- P_EXT(x, P_1, y), R(x, y, z, …), P_EXT(y, P_1, z), R(y, z, z’, …), P_EXT(z, P_1, z’), …

However, in SWIM the above constraints entail the existence of two mappings. The first constraint could imply the mapping

[19] To be more accurate, the 1st mapping rule could have additional predicates in its body. We will examine such a case in the following example.

$P\_EXT(x, P_1, y), P\_EXT(y, P_2, y') \text{ :- } R(x, y, z, y', \ldots)$

The second one implies the mapping

$P\_EXT(x, P_1, y) \text{ :- } R(z, x, y, y', \ldots)$

These mappings introduce the constraints

$\forall x \forall p \forall y \forall q \forall y'\ P\_EXT(x, p, y) \wedge p=\text{“}P_1\text{”} \wedge P\_EXT(y, q, y') \wedge q=\text{“}P_2\text{”} \rightarrow \exists z\ R(x, y, z, y', \ldots)$
$\forall x \forall p \forall y\ P\_EXT(x, p, y) \wedge p=\text{“}P_1\text{”} \rightarrow (\exists z \exists y'\ R(x, y, z, y', \ldots)) \vee (\exists z \exists y'\ R(z, x, y, y', \ldots))$
$\forall y \forall q \forall y'\ P\_EXT(x, q, y) \wedge q=\text{“}P_2\text{”} \rightarrow \exists z \exists y'\ R(z, x, y', y, \ldots)$
$\forall x \forall y \forall z \forall y'\ R(x, y, z, y', \ldots) \rightarrow \exists p \exists q\ P\_EXT(x, p, y) \wedge p=\text{“}P_1\text{”} \wedge P\_EXT(y, q, y') \wedge q=\text{“}P_2\text{”}$
$\forall x \forall y \forall z\ R(x, y, z, y', \ldots) \rightarrow \exists p \exists q\ P\_EXT(y, p, z) \wedge p=\text{“}P_1\text{”}$

Let’s see the effect of the above constraints on query

$ans(x) \text{ :- } P\_EXT(x, P_1, y)$

It will chase to the universal plan

$ans(x) \text{ :- } P\_EXT(x, P_1, y), R(x, y, z, y', \ldots), P\_EXT(y, P_2, y'), P\_EXT(y, P_1, z)$
$\cup \quad ans(x) \text{ :- } P\_EXT(x, P_1, y), R(z, x, y, y', \ldots), P\_EXT(z, P_1, x), P\_EXT(x, P_2, y')$

Once more the chase terminates. ■

Likewise, the chase with any set of constraints extracted from RDB→RDF mappings can be shown not to diverge. Thus, the following proposition is educed.

Proposition B.2: The chase with Δ_{Map} terminates. ■

The key point behind the termination of chase is the standard procedure for interpreting the mappings as constraints, and more specifically, the use of disjunction for constraints having the same head (of RVL clauses). It is the disjunction that prevents the constraints that create a cycle with an ∃-labelled edge to be applied both on the same conjunctive query.

The termination of the chase given Δ_{Map} is a very powerful condition, which encompasses cases that will rarely rise in real publishing scenarios. Consider Example

B.5. It presupposes a relation $R(a_1, a_2, a_3, \ldots)$ where both the pairs $<a_1, a_2>$ and $<a_2, a_3>$ instantiate the property P_1. Moreover, the relational attributes a_1, a_2 and a_3 must correspond to the same classes, since a_2 instantiates both the domain and range of P_1!

Appendix C

BNF Grammar for Datalog

```
[1]     PROGRAM             ::= FACTLIST RULELIST QUERY
                                ;

[2]     FACTLIST            ::= FACT FACTLIST
                                |
                                ;

[3]     FACT                ::= CLASS_FACT
                                | PROP_FACT
                                | CSUB_FACT
                                | PSUB_FACT
                                ;

[4]     CLASS_FACT          ::= “CLASS(“ CONSTANT “)” “.”
                                ;

[5]     PROP_FACT           ::= “PROP(“ TRI_CONSTANT “)” “.”
                                ;

[6]     CSUB_FACT           ::= “CSUB(“ DBL_CONSTANT “)” “.”
                                ;

[7]     PSUB_FACT           ::= “PSUB(“ DBL_CONSTANT “)” “.”
                                ;

[8]     TRI_CONSTANT        ::= CONSTANT “,” CONSTANT “,”
                                CONSTANT
                                ;

[9]     DBL_CONSTANT        ::= CONSTANT “,” CONSTANT
                                ;

[10]    RULELIST            ::= RULE RULELIST
                                |
                                ;

[11]   RULE                             ::= HEAD ":-" SQBODY
                                ;

[12]   HEAD                             ::= "CEXT(" CONSTANT "," VARIABLE
                                ")"
                                | "PEXT(" VARIABLE "," CONSTANT
                                "," VARIABLE ")"
                                ;

[13]   SQBODY                           ::= SQITEM
                                | SQITEM "," SQBODY
                                ;

[14]   SQITEM                           ::= RELATION
                                | COMPAR
                                | FUNCTION
                                ;

[15]   RELATION                         ::= CONSTANT "(" REL_LIST ")"
                                ;

[16]   REL_LIST                         ::= VARIABLE
                                | VALUE
                                | VARIABLE "," REL_LIST
                                | VALUE "," REL_LIST
                                ;

[17]   FUNCTION                         ::= CONCAT_F
                                ;

[18]   CONCAT_F                         ::= "myConcat(" VARIABLE ","
                                            MIXED_CONCAT ","
                                            MIXED_CONCAT ")"
                                            ;

[19]   MIXED_CONCAT                     ::= VARIABLE
                                | VALUE
                                ;
```

```
[20]   REL_LIST                    ::= VARIABLE
                                       | VALUE
                                       | VARIABLE "," REL_LIST
                                       | VALUE "," REL_LIST
                                       ;

[21]   VALUE                       ::= STRING_VALUE
                                       | NUMBER
                                       ;

[22]   QUERY                       ::= Q_HEAD ":-" Q_BODY "."
                                       ;

[23]   Q_HEAD                      ::= "QUERY(" VARIABLE_LIST ")"
                                       ;

[24]   VARIABLE_LIST               ::= VARIABLE
                                       | VARIABLE "," VARIABLE_LIST
                                       ;

[25]   Q_BODY                      ::= QITEM
                                       | QITEM "," Q_BODY
                                       ;

[26]   QITEM                       ::= COMPAR
                                       | "PROP(" TRP_PROP ")"
                                       | "C_SUB(" DBL_CSUB ")"
                                       | "C_EXT(" DBL_CEXT ")"
                                       | "P_SUB(" DBL_PSUB ")"
                                       | "P_EXT(" TRP_PEXT ")"
                                       ;

[27]   COMPAR                      ::= EQUALITY
                                       ;

[28]   EQUALITY                    ::= VARIABLE "=" VALUE
                                       | VARIABLE "=" VARIABLE
                                       ;

[29]   TRP_PROP                    ::= VARIABLE "," CONSTANT ","
```

```
                                          VARIABLE
                                          ;

[30]    DBL_CSUB                      ::= VARCONST “,” VARCONST
                                          ;

[31]    DBL_CEXT                      ::= VARCONST “,” VARIABLE
                                          ;

[32]    DBL_PSUB                      ::= VARCONST “,” VARCONST
                                          ;

[33]    TRP_PEXT                      ::= VARIABLE “,” VARCONST “,”
                                          VARIABLE
                                          ;

[34]    VARCONST                      ::= VARIABLE
                                          | CONSTANT
                                          ;
```