University of Crete
Computer Science Department

Integrating XML data sources using RDF/S Schemas: The ICS-FORTH Semantic Web Integration Middleware (SWIM)

Ioanna Koffina

Master's Thesis

Heraklion, February 2005

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΚΑΙ ΤΕΧΝΟΛΟΓΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

Σημασιολογική Διαμεσολάβηση Ετερογενών Πηγών Δεδομένων XML στον Σημασιολογικό Ιστό

Εργασία που υποβλήθηκε απο την

Ιωάννα Α. Κοφφινά

ως μερική εκπλήρωση των απαιτήσεων για την απόκτηση
ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΕΙΔΙΚΕΥΣΗΣ

Συγγραφέας:

Ιωάννα Κοφφινά, Τμήμα Επιστήμης Υπολογιστών

Εισηγητική Επιτροπή:

Βασίλης Χριστοφίδης, Αναπληρωτής καθηγητής, Επόπτης

Δημήτρης Πλεξουσάκης, Αναπληρωτής καθηγητής, Μέλος

Γρηγόρης Αντωνίου, Καθηγητής, Μέλος

Δεκτή:

Δημήτρης Πλεξουσάκης, Αναπληρωτής καθηγητής
Πρόεδρος Επιτροπής Μεταπτυχιακών Σπουδών

Ηράκλειο, Φεβρουάριος 2005

Integrating XML data sources using RDF/S Schemas: The ICS-FORTH Semantic Web Integration Middleware (SWIM)

Ioanna Koffina

Master's Thesis

Computer Science Department, University of Crete

Abstract

One of the main objectives of the Semantic Web (SW) is to enable interoperability of data published on the Web. The vast majority of these data reside in legacy sources, which are published on the so-called Deep Web. These sources may vary from relational database systems (with or without XML views) to native XML sources, contain useful information and can be manipulated by a plethora of query languages and searching interfaces.

In order to take advantage of these diverse sources we need a SW middleware capable of integrating data residing in these sources by employing rich mediation schemata. In particular, mediation schema should be expressed in terms of a SW language (e.g., RDF/S, OWL) capturing data semantics. The functionality of a SWIM relies on rendering source heterogeneity transparent to the user and facilitating him to query the mediated schema with declarative languages. Additionally, it should support sound and complete algorithms for query reformulation, from the mediated schema to the underlying data sources.

In this context, we propose the ICS-FORTH SWIM as a middleware for integrating XML sources by employing domain or application specific RDF/S schemas. SWIM enables users to query the virtual mediated RDF/S schema using a declarative language (e.g., RQL) and moreover, it provides further levels of abstraction using view

definition languages (e.g., RVL). The use of a well-founded fragment of first-order logic for specifying the mappings between the RDF/S and XML data models exploits the background theory on relational query reformulation and optimization. SWIM adopts a hybrid approach for establishing the mappings between the XML data sources and the mediated RDF/S schema, which is called GLAV.

Supervisor: Vassilis Christophides
Associate Professor

Σημασιολογική Διαμεσολάβηση Ετερογενών Πηγών Δεδομένων XML στον Σημασιολογικό Ιστό

Ιωάννα Κοφφινά

Μεταπτυχιακή Εργασία

Τμήμα Επιστήμης Υπολογιστών, Πανεπιστήμιο Κρήτης

Περίληψη

Ένας από τους βασικούς στόχους του Σημασιολογικού Ιστού είναι η επίτευξη της διαλειτουργικότητας των δεδομένων που δημοσιεύονται στον Παγκόσμιο Ιστό. Στην πλειονότητά τους τα δεδομένα αυτά είναι αποθηκευμένα σε παραδοσιακές πηγές και αποτελούν αυτό που αποκαλούμε 'κρυφό' παγκόσμιο ιστό. Οι πηγές αυτές ποικίλουν από σχεσιακές βάσεις δεδομένων (που μπορεί να διαθέτουν ή όχι όψεις XML) έως πηγές με πραγματικά XML δεδομένα. Σε αυτές περιέχονται χρήσιμες πληροφορίες τις οποίες διαχειρίζεται ένα πλήθος γλωσσών επερωτήσεων και διεπαφών αναζήτησης.

Προκειμένου να εκμεταλλευτούμε την διαφορετικότητα αυτών των πηγών χρειαζόμαστε ένα διαμεσολαβητή που επιτυγχάνει την ολοκλήρωση τους με την βοήθεια ενός ενδιάμεσου σχήματος. Αυτό το σχήμα πρέπει να είναι ικανό να εκμεταλλευτεί τα πλεονεκτήματα μιας γλώσσας του σημασιολογικού ιστού (όπως η RDF/S ή η OWL) στην έκφραση της σημασιολογίας των δεδομένων. Η λειτουργικότητα ενός Διαμεσολαβητή Ολοκλήρωσης στον Σημασιολογικό Ιστό (SWIM) έγκειται στο να καταστήσει την ετερογένεια των πηγών αδιαφανή στον χρήστη και να τον διευκολύνει στην υποβολή ερωτήσεων προς αυτές με την βοήθεια μιας δηλωτικής γλώσσας επερωτήσεων. Επιπλέον, πρέπει να παρέχει μηχανισμούς για την επαναδιατύπωση των επερωτήσεων από το σχήμα του διαμεσολαβητή στο σχήμα των πηγών, οι οποίοι να εγγυώνται την ορθότητα και την πληρότητα των επαναδιατυπωμένων επερωτήσεων.

Λαμβάνοντας υπόψιν τα παραπάνω προτείνουμε έναν διαμεσολαβητή ολοκλήρωσης σημασιολογικού ιστού (ICS-FORTH SWIM) ο οποίος ολοκληρώνει πηγές XML δεδο-

μένων χρησιμοποιώντας RDF/S σχήματα που περιγράφουν συγκεκριμένες εφαρμογές ή επιστημονικά πεδία. Το SWIM διευκολύνει τους χρήστες να επερωτήσουν το ενδιάμεσο ιδεατό RDF/S σχήμα με την βοήθεια δηλωτικών γλωσσών, όπως η RQL, και επιπλέον προσφέρει μηχανισμούς αφαίρεσης με την βοήθεια γλωσσών ορισμού όψεων, όπως η RVL. Η χρήση ενός καλά θεμελιωμένου υποσυνόλου της λογικής πρώτης τάξης για τον ορισμό αντιστοιχήσεων μεταξύ του RDF/S σχήματος και των XML δεδομένων, εκμεταλλεύεται προηγούμενα αποτελέσματα στην επαναδιατύπωση και βελτιστοποίηση σχεσιακών ερωτήσεων. Για την αντιστοίχηση των πηγών XML στο σχήμα διαμεσολάβησης RDF/S το SWIM χρησιμοποιεί μια υβριδική προσέγγιση η οποία ονομάζεται GLAV.

Επόπτης Καθηγητής: Βασίλης Χριστοφίδης
Αναπληρωτής Καθηγητής

Στον Γιώργο, στην Κατερίνα

και σε όσους πίστεψαν σε αυτήν την προσπάθεια

Ευχαριστίες

Στο σημείο αυτό θα ήθελα να ευχαριστήσω τον επόπτη μου, κ. Βασίλη Χριστοφίδη, για τα όσα μου προσέφερε αυτά τα δύο χρόνια της συνεργασίας μας και τις ευκαιρίες που απλόχερα μου έδωσε. Η βοήθεια του ήταν ανεκτίμητη και κοντά του αποκόμισα πολύτιμες γνώσεις. Ήταν πολύ σημαντικό για μένα το γεγονός ότι το γραφείο του ήταν ανοικτό και ο ίδιος πάντα διαθέσιμος για συζήτηση. Η επιμονή του στο να μην αφήνω τίποτα ανεπιβεβαίωτο ήταν ιδιαίτερα γόνιμη και είναι από τα σπουδαιότερα πράγματα που έμαθα κατά την διάρκεια της μεταπτυχιακής μου εργασίας. Χωρίς την ουσιαστική καθοδήγησή του και τις επισημάνσεις του η ολοκλήρωση αυτής της εργασίας θα ήταν αδύνατη. Θέλω να ελπίζω ότι στάθηκα αντάξια των προσδοκιών του.

Ιδιαίτερες ευχαριστίες αξίζουν και στον καθηγητή μου, κ. Δημήτρη Πλεξουσάκη, με τον οποίο συνεργάστηκα αρκετά χρόνια. Τον ευχαριστώ γιατί ήταν πάντα διαθέσιμος για συζήτηση και είχε πάντα να μου δώσει χρήσιμες συμβουλές κατά την εκπόνηση και εξέταση της εργασίας αυτής.

Θα ήθελα ακόμα να ευχαριστήσω και τον κ. Γρηγόρη Αντωνίου, ένα ακόμα μέλος της επιτροπής εξέτασης της μεταπτυχιακής μου εργασίας, για τις πολύτιμες παρατηρήσεις του.

I would like to thank Professor Val Tannen for his contribution in the apprehension of 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 always more than willing to answer them.

Ιδιαίτερα θα ήθελα να ευχαριστήσω το Πανεπιστήμιο Κρήτης και την ομάδα Πληροφοριακών Συστημάτων του Ινστιτούτου Πληροφορικής για τα όσα μου προσέφεραν όλα αυτά τα χρόνια και για τις γνώσεις που απέκτησα κατά τις σπουδές μου στην Κρήτη.

Ωφείλω επίσης στο σημείο αυτό να ευχαριστήσω τον Γιώργο Κοκκινίδη που επί έξι χρόνια στέκεται πλάι μου και με στηρίζει με όλες του τις δυνάμεις. Οι προσπάθειες του να κατανοήσει τις παραξενιές μου και να καθησυχάσει το άγχος μου ήταν, για μένα, κάτι παραπάνω από σημαντικές.

Επίσης, θα ήθελα να ευχαριστήσω όλους τους φίλους μου και ιδιαίτερα αυτούς των φοιτητικών μου χρόνων (Βάσω, Γιώργο, Δέσποινα, Νίκο και Παναγιώτη) για όσα

περάσαμε μαζί αλλά και για το ότι πίστεψαν σε μένα. Ιδιαίτερες ευχαριστίες αξίζουν στον Γιώργο για την επί δύο χρόνια στενή συνεργασία μας.

Το μεγαλύτερο όμως ευχαριστώ το αξίζουν οι γονείς μου, Τάσος και Ρούλα, και η αδερφή μου, Κατερίνα, για τα όσα θυσίασαν για μένα, για την ακούραστη στήριξη τους, αλλά κυρίως για το ότι πίστευαν ότι μπορώ να τα καταφέρω. Σας ευχαριστώ πολύ για όλα!

Ιωάννα Κοφφινά

Contents

1 Introduction **1**

1.1 A Motivating Example 2

1.2 Contributions 4

1.2.1 A Formal Framework for Mapping Specification 5

1.2.2 Query Reformulation and Optimization 5

1.3 Organization of the document 6

2 Introduction to XML and RDF **9**

2.1 XML 9

2.1.1 XML DTDs and Schemas 11

2.1.2 XPath 15

2.1.3 XQuery 18

2.2 RDF 21

2.2.1 RDF Query Language (RQL) 25

2.2.2 A View Definition Language for RDF (RVL) 27

2.3 Comparison of XML and RDF 29

2.4 Conclusion 30

3 Data Integration Systems **31**

3.1 Data Integration Architectures 33

3.1.1 Virtual View Approach 33

3.1.1.1 Federated Database Systems (Read/Write Systems) . 33

3.1.1.2 Mediated Systems (Read Only Systems) 35

3.1.2 Materialized View Approach 36
3.2 Modelling the Data Sources 37
3.2.1 Global Centric Approach 39
3.2.2 Local Centric Approach 43
3.2.3 Query Reformulation Algorithms 45
3.2.3.1 The Bucket Algorithm 45
3.2.3.2 Inverse Rules Algorithm 47
3.2.3.3 MiniCon Algorithm 49
3.2.3.4 The Shared Variable Bucket Algorithm 50
3.2.4 Combining Global and Local Approach 50
3.2.4.1 Another Hybrid Approach: BAV 52
3.2.5 Comparison of the Approaches 53
3.2.6 Representative Systems 54
3.2.6.1 GAV Systems 54
3.2.6.2 LAV Systems 57
3.2.6.3 GLAV Systems 58

4 The SWIM Logic Framework 61
4.1 Preliminary Definitions 61
4.2 The First-Order Logic Representation of XML 62
4.2.1 XML Basic Relations 62
4.2.2 XML Basic Constraints 66
4.3 The First-Order Logic Representation of RDF 68
4.3.1 RDF/S Basic Relations 69
4.3.2 RDF/S Basic Constraints 71
4.4 Differences with the RDF/S Model Theory 73

5 Query Reformulation in SWIM 75
5.1 From XML Data to RDF/S Schema 75
5.2 SWIM Mapping Rules 77
5.2.1 Capturing Mappings through Constraints 80

5.2.2 The XPath Fragment used for Mappings 83
5.2.3 Expressiveness of the SWIM Mappings 84
5.3 Query Reformulation Algorithm 90
5.3.1 Chase & Backchase Algorithms 90
5.3.2 SWIM's Query Reformulation Algorithm 94
5.4 Improving Reformulations with XML Data Semantics 104
5.5 Query Translation 108
5.5.1 The XPath Translation 110
5.5.2 The XQuery Translation 111
5.6 How SWIM Addresses Heterogeneity Issues 112
5.6.1 Syntactic Heterogeneity 113
5.6.2 Structural Heterogeneity 114
5.6.3 Semantic Heterogeneity 116

6 SWIM Architecture 121
6.1 SWIM Query Reformulation Engine 122
6.2 SWLF Compiler 124
6.3 MARS System: A chase/backchase machine 124
6.4 XPath/XQuery Generator 125
6.5 Related Work 126
6.5.1 Styx 126
6.5.2 PEPSINT 126
6.5.3 Clio 127
6.5.4 Piazza 128
6.5.5 RDF Based Architecture for Semantic Integration of Heterogeneous Information Sources 129
6.5.6 YACOB 129
6.5.7 Integration of XML sources with OWL 130
6.5.8 WEESA - WEb Engineering for Semantic web Applications . 130
6.5.9 Interpreting XML via an RDF Schema 131

7 Conclusion and Future Work **133**

7.1 Future Work 134

Bibliography **137**

A RQL Translation Patterns **147**

B BNF grammar for Datalog Rules **151**

List of Tables

2.1 Basic differences between DTD and XML Schema 14
2.2 Simplified BNF Grammar for XPath 2.0 17
2.3 Simplified BNF Grammar for XQuery 1.0 21
2.4 Simplified BNF Grammar for RQL 26

3.1 Basic characteristics of the information integration systems 55

5.1 Simplified BNF Grammar for Mapping Rules 80
5.2 Simple XPath Grammar 83
5.3 The facts that arise from RDF/S schema 96
5.4 The mappings between XML and RDF/S 97

List of Figures

1.1 Republishing XML as RDF 3

2.1 The XPath2.0 navigational axes 16
2.2 XQuery FLWR ("Flower") Expression 19
2.3 The graph representation of RDF 22
2.4 RVL virtual schemata 28

3.1 The architecture of a federated database system 34
3.2 The architecture of a mediated integration system 35
3.3 The architecture of a data warehouse 37
3.4 Global centric approach 40
3.5 Query unfolding 42
3.6 Local centric approach 43
3.7 GLAV approach 51
3.8 The basic components of Tsimmis System 56
3.9 Information Manifold Architecture 57
3.10 Dis@Dis Architecture 59

4.1 An example of an XML tree 65
4.2 An example of an RDF schema 71

5.1 Defining correspondences between XML and RDF schema 77
5.2 Mapping interpretation to constraints 81
5.3 The tree representation of the XML structure 85
5.4 The RDF/S schema 86

5.5 Mapping an XML edge to an RDF/S property 86
5.6 Mapping an XML path to an RDF/S property 87
5.7 Mapping an XML tree to an RDF/S property 87
5.8 Mapping an XML tree to an RDF/S path 88
5.9 The correspondences between the XML and RDF/S schema of our example 89
5.10 The general problem of query reformulation 90
5.11 General C&B Algorithm 91
5.12 Chase example 91
5.13 Chase flow graph 93
5.14 Backchase example 93
5.15 The reconstruction of the reformulated query 109
5.16 Heterogeneity map 113

6.1 The ICS-FORTH SWIM Architecture 122
6.2 Architecture of the SWIM Query Reformulation Engine 123

Chapter 1

Introduction

Recent experimental studies highlight the fact that the size of the so-called *Deep Web* [Dee] (i.e., data stored in legacy databases and accessed through Web forms) largely exceeds the size of the *Surface Web* (i.e., data stored in static HTML pages). Data on the Deep Web are managed under a variety of formats (relational databases or XML) and they are queried using various searching interfaces and languages (i.e., SQL, XPath, XQuery). In this context, a key factor for the success of the Semantic Web (SW) is to provide a useful, comprehensive and high-level access to voluminous data residing in Deep Web sources.

More precisely, SW technology should be able to publish legacy data as valid instances of domain or application specific RDF/S schemas (or other SW ontology languages). In other words, we need a SW integration middleware (SWIM) capable to either republish XML as RDF, or to publish RDB data directly as RDF, or even better, capable of performing both.

There are many issues involved in the functionality of a SWIM. In particular, SWIM may act as an integrator for relational and XML sources. Such an integrator should facilitate users to formulate queries against the mediated RDF/S schema using declarative languages (such as RQL [KAC+02]), as well as, support further abstraction levels using declarative view definition languages (e.g., RVL [Mag03]). In a nutshell, SWIM should offer the following services: (a) establish mapping rules between XML and RDF and between RDB and RDF, (b) verify the conformance of these mappings

w.r.t the semantics of the employed schemas, (c) reformulate RDF/S queries against RDB or XML sources, and (d) combine these queries with RVL views.

In order to address effectively and efficiently the above requirements, we should choose a uniform and expressive logic framework to define SWIM integration services. This framework should exploit background theory on conjunctive queries and query containment and minimization [AHV95].

An architecture based on mediators is highly beneficial for deploying a SWIM. There exist two main approaches for integrating data sources [BB04] using mediator-based architectures: Global-as-View (GAV) [Ull00] and Local-as-View (LAV) [Lev99] [Lev01]. The former provides descriptions of the global schema in terms of the views of local sources and relies on simple query reformulation techniques (i.e., query unfolding). The latter considers local sources as materialized views specified in terms of the global schema. LAV supports easily the evolution of the data integration system by just adding or removing the descriptions of local sources. In our work we advocate a hybrid approach called GLAV [FLM99], which combines the previous advantages and exceeds the expressive power of both GAV and LAV.

1.1 A Motivating Example

Let's assume an XML source whose content is described by a DTD or an XML Schema (see Figure 1.1). XML data from this source contain information about Museums, exhibiting some artifacts for which we want to know their creator. Data stored in such sources can be queried using an XML query language like XPath [XPaa] [XPab] or XQuery [XQu].

Now suppose that we add on top of this repository an RDF/S schema from the cultural domain. This mediated RDF/S schema can be queried using RQL and it can be used for defining personalized views with the help of RVL. However, since there are no actual RDF data, we need to reformulate the RQL queries expressed against this virtual RDF/S schema into queries appropriate for our XML source. For example, the following RQL query:

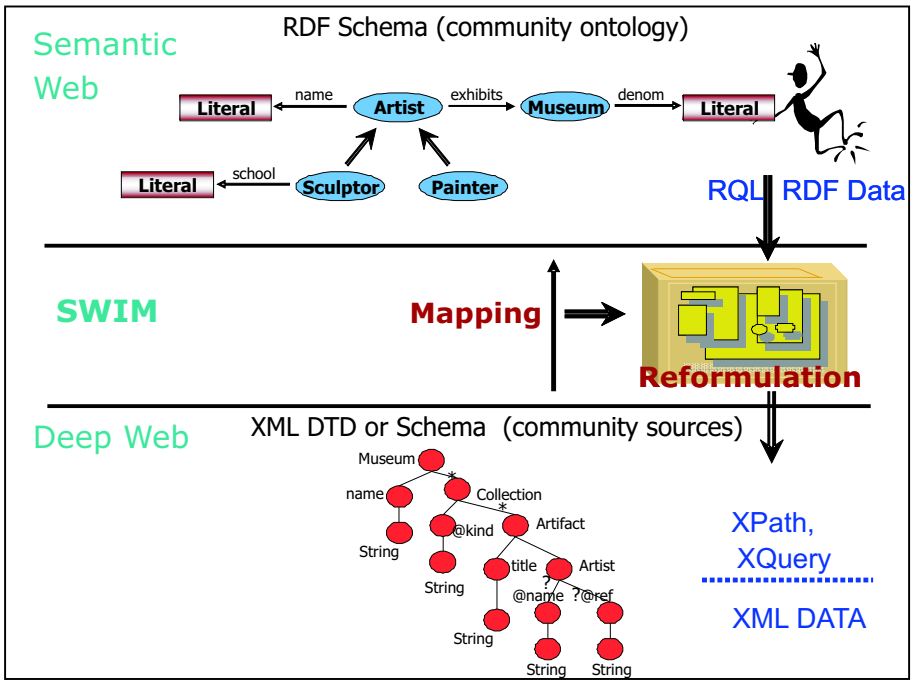


Figure 1.1: Republishing XML as RDF

SELECT X
FROM {X}exhibits{Y}, {Y}denom{Z}
WHERE Z = “Louvre”

will be *reformulated* to the following XQuery query:

```
<RDF> {
  <Bag>
  {
    for $var0 in document(“art.xml”)//Museum
    for $var1 in $var0/name
    for $var2 in $var0//Artist/@name
    where $var1/text()=“Louvre”
    return <li>{$var2/text()}</li>
  }
  </Bag>
} </RDF>
```

This reformulation involves several challenging issues. First of all, the schemas employed by our XML sources and the RDF/S mediator are different. Their discrepancies, usually called *heterogeneity conflicts*, can be classified under three axes: syntactic, structural and semantic [SK92]. As we can see in our example (Figure 1.1), we need to view the XML data through the RDF data model (syntactic conflict), to resolve categorization conflicts, given that in RDF/S there is a class hierarchy while in XML there isn't (structural conflict), as well as address naming mismatches; for instance the "name" of a Museum in XML is called "denomination" in RDF/S (semantic conflict).

In order to reconciliate the heterogeneous representations of our data we need to define appropriate *mapping rules*. Choosing an expressive but tractable logical framework to map data from XML to RDF/S is crucial for the success of a SWIM. These mappings are used for reformulating queries issued against the virtual RDF/S schema into queries acceptable by our XML sources. However, more complex mappings (in order to increase expressiveness) render query reformulation harder. Query reformulation becomes more complex if we take into consideration the presence of constraints capturing the semantics of both the RDF/S and XML data models, as well as, application-specific constraints coming from the schema (if any) of XML sources (like keys, foreign keys etc.). So, there is a need for a sound and complete reformulation algorithm.

Since reformulated queries are evaluated to remote sources and mediator queries resulting from automated manipulation/generation may entail redundancies, their optimization is crucial. In particular, optimization tries to simplify queries by removing redundant predicates and to eliminate redundant queries.

1.2 Contributions

In this context, we propose a middleware called ICS-FORTH SWIM (Semantic Web Integration Middleware) supporting the following functionalities:

1.2.1 A Formal Framework for Mapping Specification

As discussed previously, choosing a logical framework for defining the mappings is of great importance. Our idea was to represent both RDF and XML data models as first-order logic predicates and capture their semantics through appropriate constraints. In this way we reduce RDF to XML query reformulation problem to the relational equivalent one, and thus, we can reuse existing techniques for relational query containment and minimization.

More precisely, we rely on Linear Datalog (non-recursive Datalog without negation) for establishing the mappings and translating the RQL/RVL queries and views issued against the virtual RDF/S schema. The head of the Datalog rules consists of a conjunction of view clauses employed by the RDF/S view definition language RVL, and the body consists of XPath atoms that facilitate querying tree-structured XML sources. The former is used in order to point out the instantiation of RDF/S schemas with appropriate resources residing in our XML data sources. Employing some non interpreted built-in predicates (e.g., concat, split) for handling more intricate cases (like complex keys or string manipulation) enhances the expressiveness of the mappings.

As far as these mappings are concerned, they are interpreted in a constraint - oriented way implementing the GLAV approach of our middleware. We can map a *view* over the global RDF/S to a *view* over local XML sources, and each of these mappings is captured with the help of constraints. Constraints describe both the RDF/S schema in terms of the XML sources (GAV) and the XML sources in terms of the global RDF/S schema (LAV).

1.2.2 Query Reformulation and Optimization

Another powerful functionality of the SWIM is its ability to reformulate queries. RQL queries, expressed in terms of the RDF/S virtual schema, result in minimized queries expressed in terms of the XML sources by gradually applying a number of chasing/backchasing [DT03a] [DT03b] [DT02] steps. The use of this algorithm is

proven to be sound and complete for disjunctions of conjunctive queries in the presence of disjunctive embedded dependencies (DEDs). This is guaranteed by the use of the Chase/Backchase algorithm given this type of input.

A query is chased with the help of the constraints that have been defined to express the semantics of the XML and RDF data models and in addition, with constraints (if any) coming from the XML data sources i.e., specification of keys and foreign keys, as well as, of domain constraints (e.g., enumerated types). The result of chasing is backchased for producing a minimal reformulation. These minimized queries are simplified (retaining as few predicates as possible) and the redundant ones are eliminated. In this way we guarantee that we query the XML sources with the minimum possible queries. Finally, the minimized reformulated queries are translated into XPath and/or XQuery.

The work presented in this thesis has been published in the proceedings of the First International Workshop on Semantic Web and Databases (SWDB 03) [CKK+03] and has been presented in the Dagstuhl Seminar on "Semantic Interoperability and Integration" [KSC+04].

1.3 Organization of the document

This thesis is organized as follows:

Chapter 2 introduces the XML and RDF data models, along with their respective querying and viewing languages. The aim of this presentation is to focus on the special features and the expressiveness of the languages on which we will concentrate our attention in the remainder of this thesis.

Chapter 3 is a report on the state-of-the-art on data integration systems. Firstly, we analyze the basic issues concerning data integration systems and then we present the basic data integration architectures. Additionally, we discuss and compare in detail the basic approaches for modeling the data sources in a mediator-based architecture along with the basic query reformulation algorithms used by such approaches. For each approach we briefly present the most representative data integration systems.

In Chapter 4 we discuss the logical framework used by the SWIM middleware. SWIM represents both RDF/S and XML schema constructs as first-order logic predicates along with constraints used for preserving the semantics of both data models. This framework actually presents our first contribution on the specification of an expressive logical framework for determining mapping rules.

Chapter 5 presents our next contributions regarding query processing (query reformulation and optimization) performed by SWIM. Firstly, it discusses the peculiarities that appear in the XML to RDF mapping. In addition, we present the basic steps of the query reformulation algorithm and explain it through a running example. The problem of heterogeneity, through the lenses of our solution, concludes this chapter.

Chapter 6 presents the basic components of the SWIM middleware. We discuss thoroughly their functionality and we explain the choices we made for the implementation of these parts. Furthermore, we refer to the most relevant works that have been published recently, comparing them with SWIM.

Chapter 7 summarizes the work of this thesis and presents some interesting research directions for future work.

Chapter 2

Introduction to XML and RDF

In this chapter we provide an overview of both the XML and the RDF data models, as well as related query languages. The objective of this chapter is twofold. On the one hand to detail the constructs of the two data models involved in our thesis and on the other hand to highlight their differences in representing and manipulating data.

2.1 XML

Extensible Markup Language (XML) is a simple, very flexible text format derived from SGML. While XML was originally designed to meet the challenges of large-scale electronic publishing, it plays an increasingly important role in the exchange of a wide variety of data on the Web [XMLa] [ABS00].

XML is an extremely versatile data format that has been used to represent many different kinds of data, including web pages, books, XML representations of relational database tables, programming interfaces, objects and multimedia presentations. In addition, some systems offer XML views of non-XML data sources such as relational databases, allowing XML-based processing of data that are not physically stored as XML.

The XML data model is a tree with labeled and ordered nodes. This means that the order of each node is meaningful and important for writing an XML document.

XML is a markup language like HTML. However, XML is designed to *describe*

data and to focus on *what data is*, while, HTML is designed to *display data* and to focus on *how data looks*. So, XML is not a replacement for HTML since they have different goals.

A common characteristic of XML and HTML is the use of tags. The tags used to mark up HTML documents and their structure are predefined. The author of an HTML document can only use tags that are defined in the HTML standard (like <p>, <h1>, etc.). However, XML tags are not predefined. XML allows the author to define his own tags and his own document structure.

For example, an XML document looks like:

```
<Art>
    <Artist id="001">
        <Name>Pablo Picasso</Name>
        <Born>25 October 1881</Born>
        <Artifacts>
            <Title>Guernica</Title>
            <Title>Las Meninas</Title>
        </Artifacts>
    </Artist>
</Art>
```

The tags, which appear in this example can be meaningful for humans, since they are user defined. This XML document can give to a (human) user information about artists and the artifacts that they have created, in a quick glance.

This example, according to the XML data model, is a tree that has nodes with labels `Art`, `Artist`, `Name`, `Born`, etc. Furthermore, the node labeled `Name` must precede the one labeled `Born`, node `Born` must precede the node labeled `Artifacts`, etc.

XML differs from HTML in three major respects:

1. New tags may be defined.

2. Structures can be nested to arbitrary depth.

3. Optional description of its grammar can be contained.

2.1.1 XML DTDs and Schemas

There are many grammar languages that can describe the structure of an XML document. The most common are: DTD and XML Schema. XML documents can be either *well-formed* or *valid*. A well-formed document is one that has matching markups, that is, it is syntactically correct. A valid XML document is one that has a schema (DTD or XML Schema) and conforms to it. A valid XML document is, of course, and a well-formed one.

DTD [DTD] stands for Document Type Definition and serves as a grammar for an underlying XML document [ABS00]. It is precisely a context-free grammar for the document. The basic components of a DTD grammar are elements, attributes and entities. A simple DTD for the XML document presented above is:

```
<!DOCTYPE Art [
    <!ELEMENT Art(Artist)*>
    <!ELEMENT Artist(Name, Born, Artifacts?)>
    <!ELEMENT Name (#PCDATA)>
    <!ELEMENT Born (#PCDATA)>
    <!ELEMENT Artifacts(Title)*>
    <!ELEMENT Title (#PCDATA)>
    <!ATTLIST Artist id ID #REQUIRED>
]>
```

In this DTD example we can mention some of the basic components: elements and attributes. Declarations starting with `ELEMENT` indicate an element which can be a complex element (has further structure, e.g., `Artist`) or a simple one (atomic type element e.g., `Name`). Occurrence indicators (if any) have the following meaning: "`?`" stands for an optional element, "`*`" stands for zero or more appearances of the element and "`+`" stands for one or more appearances of the element. The absence of an indicator stands for exactly one appearance of the element. Declarations starting with

ATTLIST indicate an attribute which can be either optional (#IMPLIED) or mandatory (#REQUIRED). There can exist some special attributes like ID or IDREF, which identify a specific element or refer to a specific element respectively. An entity (there is no entity in our example) is a symbolic name. It can represent text, symbols, or externally defined information.

One limitation of the DTD is that it imposes order. This is a restriction, since in most data applications, the ordering of the different elements, not only is unimportant but also decreases efficiency. Additionally, there is no variety of atomic types. We cannot declare that Born is of type "Date". The only atomic type that is supported is PCDATA, which means "String". In DTDs the types that are associated with the tags are global. This means that we cannot define an element "Born" to be both of types "Date" and "String". Another serious limitation is that DTDs do not constrain the type of IDREFs. They are just syntax. From the discussion above, it is clear that DTDs are somehow unsatisfactory, and several proposals have been made after the adoption of the XML standard for better schema formalisms.

A more expressive schema formalism is the XML Schema. The W3C XML Schema Definition Language [XMLb] is an XML language for describing and constraining the content of XML documents. W3C XML Schema is a W3C Recommendation. The basic characteristics of the XML Schema is that it can define elements and attributes that can appear in a document, define which elements are child elements, the order of these elements and their cardinality.

The XML Schema for the XML document above is:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

 <xsd:element name="Art" type="ArtType"/>

 <xsd:complexType name="ArtType">
  <xsd:sequence>
    <xsd:element name="Artist" type="ArtistType"
                    minOccurs="0" maxOccurs="unbounded"/>
```

```
  </xsd:sequence>
 </xsd:complexType>

 <xsd:complexType name="ArtistType">
  <xsd:sequence>
    <xsd:element name="Name" type="xsd:string"/>
    <xsd:element name="Born" type="xsd:date"/>
    <xsd:element name="Artifacts" type="ArtifactType" minOccurs="0"/>
  </xsd:sequence>
  <xsd:attribute name="id" type="xs:id"/>
 </xsd:complexType>

 <xsd:complexType name="ArtifactType">
  <xsd:sequence>
     <xsd:element name="Title" type="xsd:string"
                          minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
 </xsd:complexType>
</xsd:schema>
```

The basic constructs of an XML schema are shown in this example. In this schema the user can define complex or simple elements (depending on whether they have further structure on not) and cardinalities for them. Additionally, XML Schema gives to the user the ability to define some integrity constraints (ICs) like keys and keyrefs (foreign keys). This declaration, and any other declaration, is not necessarily global. XML Schema provides user with the ability to define an element or an attribute into a specific scope. Another useful feature is the use of namespaces. This means that a schema can be viewed as a collection (vocabulary) of type definitions and element declarations whose names belong to a particular namespace called a target namespace. Target namespaces enable us to distinguish between definitions and declarations from different vocabularies.

Features	DTD	XML Schema
Syntax in XML	No	Yes
Supporting namespace	No	Yes
include & import	No	Yes
Built-in types	10	45
User defined types	No	Yes
Type domain constraints	No	Yes
Explicit null value	No	Yes
Type extension	No	Yes except simple type
Attribute default value	Yes	Yes
Choice among attributes	No	No
Optional/required attributes	Yes	Yes
Attribute domain constraints	Partial	Yes
Element default value	No	Partial
Element content model	Yes	Yes
Choice among elements	Yes	Yes
Min & Max Occurrence	Partial	Yes
Unordered list	No	Yes
Uniqueness for attributes	Yes	Yes
Uniqueness for non-attributes	No	Yes
Key for attributes	Yes	Yes
Key for non-attributes	No	Yes
Foreign Key for attributes	Partial	Yes
Foreign Key for non attributes	No	Yes

Table 2.1: Basic differences between DTD and XML Schema

XML Schema is believed to be a replacement for DTD since it is extensible to future additions, richer and more useful than DTDs.

One of the basic differences between the DTDs and the XML Schemas is that the first provides grammar for an XML document giving the basic "shape" of the document. The XML Schema provides this, plus a detailed way to define what the data can and cannot contain. Additionally, XML Schema supports reusability, due to the ability to use namespaces and import other schemas.

As far as the supporting types of the two grammars, XML Schema supports much more built-in types and gives the ability for user-defined ones. DTD does not have such option and supports only 10 build-in types. These differences and some more are summarized in Table 2.1.

2.1.2 XPath

XPath is a non-XML language for identifying parts of XML documents. XPath is a major element in XPointer and XSLT and it is a W3C Standard.

XPath 2.0 [XPab] is an expression language that allows processing of values conforming to its data model. This data model provides a tree representation of XML documents, as well as, atomic values (such as integers, strings, and booleans) and sequences that may contain both references to nodes in an XML document and atomic values. The result of an XPath expression may be a selection of nodes from the input documents, or an atomic value, or more generally, any sequence allowed by the data model. The name of the language derives from its most distinctive feature, *the path expression*, which provides a means of hierarchic addressing of the nodes in an XML tree. XPath 2.0 is a superset of XPath 1.0 [XPaa], with the added capability of supporting a richer set of data types, and taking advantage of the type information that becomes available when documents are validated using XML Schema.

Given the simple XML document used throughout this section, the XPath expression below selects the `Artist` elements of the document:

```
/Art/Artist
```

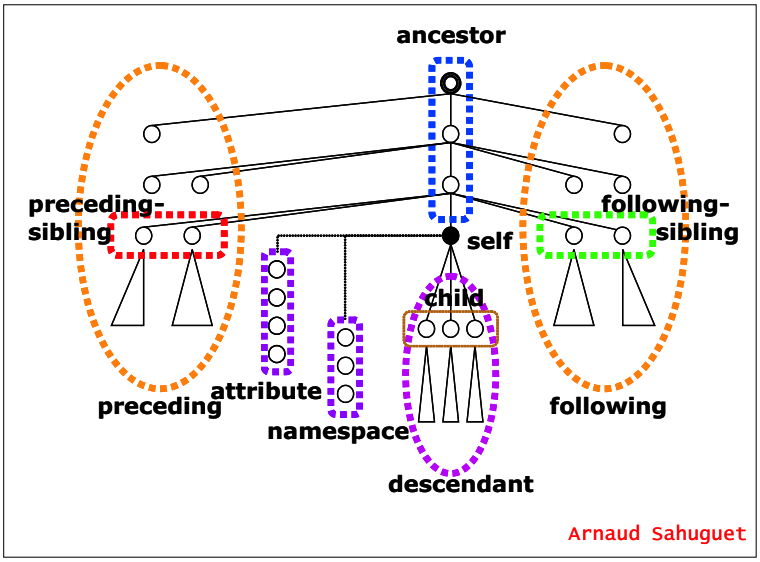


Figure 2.1: The XPath2.0 navigational axes

while the XPath expression below selects the `Titles` of the `Artifacts` that have been created by an `Artist` whose `Name` is "Pablo Picasso":

```
/Art/Artist[Name=''Pablo Picasso'']//Title
```

A simplified BNF grammar for the XPath 2.0 is shown in Table 2.2. This is not the full grammar of XPath 2.0, since it has many features not mentioned here. The most basic additional features are:

- For loop expression
- If-Then-Else conditions
- Schema testing
- Functions
- Quantified expressions
- Processing Instructions
- "And" or "Or" expressions
- Equalities or inequalities

[1]	XPath	::=	PathExpr
[2]	PathExpr	::=	*"/"* RelativePathExpr?
			| *"//"* RelativePathExpr
			| RelativePathExpr
[3]	RelativePathExpr	::=	StepExpr ((*"/"* | *"//"*) StepExpr*
[4]	StepExpr	::=	AxisStep
			| FilterExpr
[5]	AxisStep	::=	ForwardStep PredicateList
			| ReverseStep PredicateList
[6]	ForwardStep	::=	ForwardAxis NodeTest
			| AbbrevForwardStep
[7]	ForwardAxis	::=	*"child" "::"*
			| *""descendant" "::"*
			| *"attribute" "::"*
			| *"self" "::"*
			| *"descendant-or-self" "::"*
			| *"following-sibling" "::"*
			| *"following" "::"*
			| *"namespace" "::"*
[8]	AbbrevForwardStep	::=	*"@"*? NodeTest
[9]	ReverseStep	::=	ReverseAxis NodeTest
			| AbbrevReverseStep
[10]	ReverseAxis	::=	*"parent" "::"*
			| *"ancestor" "::"*
			| *"preceding-sibling" "::"*
			| *"preceding" "::"*
			| ‘‘*ancestor-or-self" "::"*
[11]	AbbrevReverseStep	::=	*".."*
[12]	NodeTest	::=	TextTest
			| NameTest
[13]	NameTest	::=	QName
			| Wildcard
[14]	Wildcard	::=	*"*"*
[15]	FilterExpr	::=	PredicateList
[16]	PredicateList	::=	(Predicate)*
[17]	Predicate	::=	*"["* PathExpr *"]"*
[18]	TextTest	::=	*"text" "()"*
[19]	QName	::=	Qualified Names

Table 2.2: Simplified BNF Grammar for XPath 2.0

A brief analysis of the BNF grammar presented, shows that it models the simple (and most common) XPath expressions used also in the previous examples. The basic components are the axes (forward or reverse) that are used for navigation into the XML document. These axes are depicted in Figure 2.1. The navigation can be either with a "full" step (e.g. Artist/child::Name) or with an abbreviated step (e.g. Artist/Name) (rules 2-11). Additionally, this grammar is able to express predicates of the form [path] (e.g Artist[Artifact/Title]). Predicates of the form [path=value] are also supported by the XPath 2.0 but are not shown in this particular fragment of the grammar. Qualified Names (rule 19) are either simple (attribute or element) names or (attribute and element) names prefixed with a namespace.

Although XPath 2.0 has many additional features compared to XPath 1.0, it also has some limitations. The basic one is that it returns XML tree nodes and not an XML document. This is a shortcoming, since querying (or navigating) an XML document usually demands an XML output. The ability of producing and moreover restructuring an XML document is valuable and is offered by the XQuery, discussed below.

2.1.3 XQuery

As an increasing amount of information is stored, exchanged, and presented using XML, the ability to query XML data sources in an efficient way becomes increasingly important. One of the great strengths of XML is its flexibility in representing many different kinds of information from diverse sources. To exploit this flexibility, an XML query language must provide features for retrieving and interpreting information from these diverse sources.

XQuery ([XQu] [CDF+03]) is designed to meet the requirements identified by the W3C XML Query Working Group. It is created to be a language in which queries are concise and easily understood. The requirement was for both a human-readable query syntax and an XML-based query syntax, and XQuery is designed to meet the first one. XQuery is derived from an XML query language called Quilt, which in turn borrowed features from several other languages, including XPath 1.0, XQL, XML-QL,

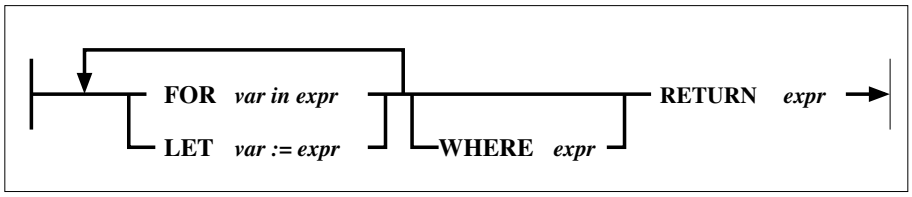


Figure 2.2: XQuery FLWR ("Flower") Expression

SQL, and OQL.

XQuery operates on the abstract, logical structure of an XML document, rather than its surface syntax. This logical structure is known as the data model.

XQuery Version 1.0 is an extension of XPath Version 2.0. Any expression that is syntactically valid and executes successfully in both XPath 2.0 and XQuery 1.0 will return the same result in both languages. Since these languages are so closely related, their grammars and language descriptions are generated from a common source to ensure consistency.

The type system of XQuery is one of the most eclectic, unusual, and useful aspects of the language [CDF+03]. XML documents contain a wide range of type information, from very loosely typed information without even a DTD to rigidly structured data corresponding to relational data or objects. A language designed for processing XML must be able to deal with this fact gracefully; it must avoid imposing assumptions on what is allowed that conflict with what is actually found in the data, allow data to be managed without forcing the programmer to cast values frequently, and allow programmer to focus on the documents being processed and the task to be performed rather than quirks of the type system. The type system of XQuery is based on XML Schema, discussed previously.

Queries in XQuery often combine information from one or more sources and restructure it to create a new result. The expression that is most commonly used for combining and restructuring the XML details is called FWLR expression. Figure 2.2 shows the general form of a FLWR expression (pronounced "flower" expression). These expressions are similar to the SELECT-FROM-WHERE statement in SQL, but they bind variables to values in `for` and `let` clauses and use these variable bindings

to create new results.

FLWR is an acronym standing for the first letter of the clauses that may occur in a FLWR expression:

- `for` clauses: associate one or more variables to expressions, creating a tuple stream in which each tuple binds a given variable to one of the items to which its associated expression evaluates
- `let` clauses: bind variables to the entire result of an expression, adding these bindings to the tuples generated by `for` clause, or creating a single tuple to contain these bindings if there is no `for` clause
- `where` clauses: filter tuples, retaining only those tuples that satisfy a condition
- `return` clauses: build the result of the FLWR expression for a given tuple

As shown in Figure 2.2, a FLWR expression starts with a sequence of `for` and `let` clauses in any order, followed by an optional `where` clause and a required `return` clause.

The simplified BNF for the XQuery queries is shown in Table 2.3.

The power of the XQuery does not stop here. XQuery provides the user with the ability to use built-in functions (like `min()`, `max()`, `average()` etc.) or functions defined by himself (user-defined functions). This makes queries simpler to be understood, and functions can be reused in other (parts of) queries. In addition, XQuery offers two sets of types: built-in types (available in any query) and types imported into a query from a specific schema.

An example of an XQuery, is the query below that returns the `names` of the `artists` that have created at least one `artifact`.

```
for $a in doc("Art.xml")//Artist
where exists($a/Artifacts/Title)
return
    <result> {$a/Name/text()} </result>
```

Summarizing, XML's popularity is based on the fact that it is a simple, human-readable language and really flexible. In XML you can represent a piece of information

[1]	QueryBody	::=	Expr
[2]	Expr	::=	ExprSingle (*","* ExprSingle)*
[3]	ExprSingle	::=	FLWORExpr
			| QuantifiedExpr
			| IfExpr
			| OrExpr
[4]	FLWORExpr	::=	(ForClause | LetClause)+ WhereClause?
			OrderByClause? *"return"* ExprSingle
[5]	ForClause	::=	*"for $"* VarName *"in"* ExprSingle
			(*", $"* VarName *"in"* ExprSingle)*
[6]	LetClause	::=	*"let $"* VarName *":="* ExprSingle
			(*", $"* VarName *":="* ExprSingle)*
[7]	WhereClause	::=	*"where"* Expr
[8]	OrderByClause	::=	*"order by"* OrderSpecList
[9]	OrderSpecList	::=	OrderSpec (*","* OrderSpec)*
[10]	OrderSpec	::=	ExprSingle OrderModifier
[11]	OrderModifier	::=	(*"ascending"* | *"descending"*)?
			(*"empty greatest"* | *"empty least"*)?
[12]	QuantifiedExpr	::=	(*"some $"* | *"every $"*) VarName *"in"* ExprSingle
			(*", $"* VarName *"in"* ExprSingle)*
[13]	IfExpr	::=	*"if ("* Expr *") then"* ExprSingle *"else"* ExprSingle
[14]	OrExpr	::=	XPath (Table 2.2) (*"or"* XPath (Table 2.2))*
[15]	VarName	::=	Qualified Name

Table 2.3: Simplified BNF Grammar for XQuery 1.0

in almost any way you want. Additionally, it offers increasing typing precision in XML specifications. There are well-formed XML documents that are already better than plain text and valid XML documents, whose structure conforms to a DTD or to an XML Schema.

2.2 RDF

The Web provides a simple and universal infrastructure to exchange various kinds of information. In order to share, interpret, and manipulate information worldwide, the role of *metadata* is widely recognized. Indeed, metadata allow us to easily locate information available in the Web, by providing descriptions about the structure and the content of the various Web resources (e.g., data, documents, images, etc.) and for different purposes. The emergence of the Resource Description Framework (RDF)

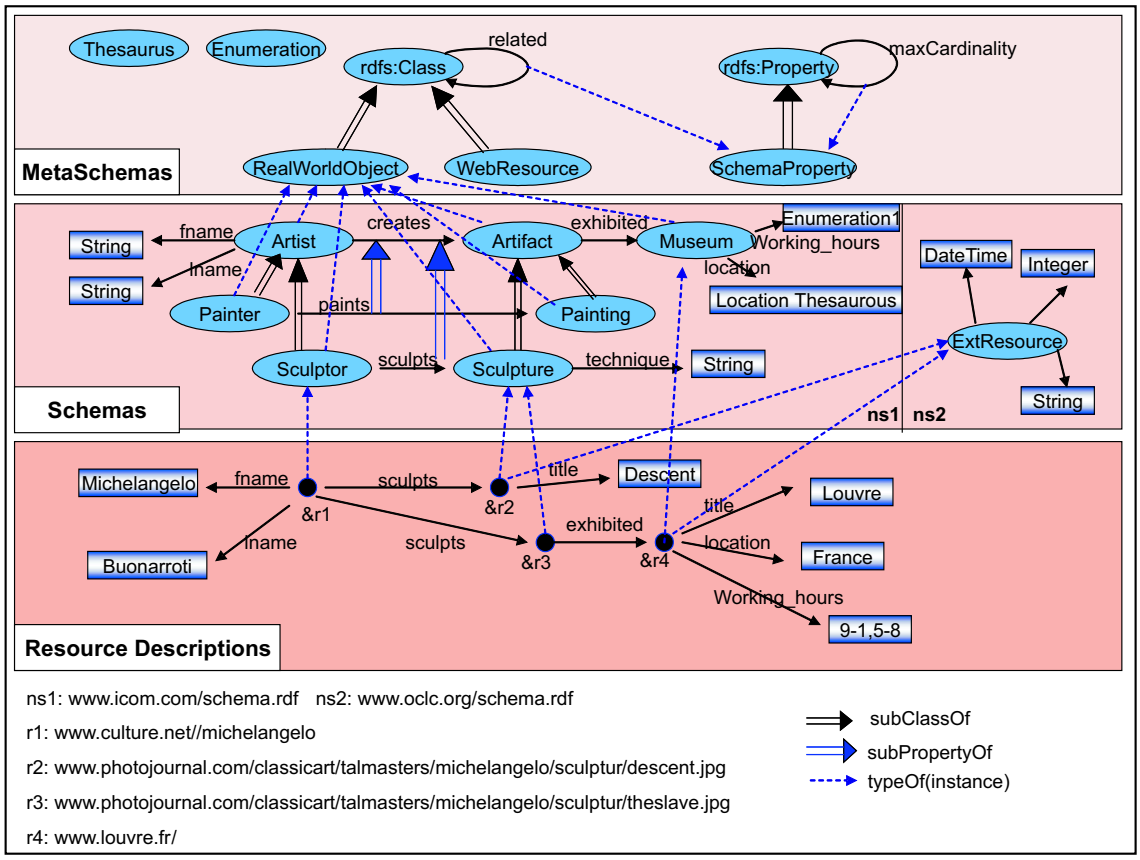


Figure 2.3: The graph representation of RDF

[rdfa] [rdfb] was expected to enable metadata interoperability across different communities or applications by supporting common conventions about metadata syntax, structure, and semantics.

More precisely, it provides (i) a Standard Representation Language for Web metadata; and (ii) a Schema Definition Language (RDF/S) to interpret (meta)data using specific class and property hierarchies (i.e., vocabularies). Moreover, RDF(/S) offer a syntax for representing metadata and schemas in XML, enabling the creation and the exchange of RDF descriptions in a both human readable and machine understandable form.

RDF is based on a directed graph model that implies the semantics of resource description. The basic idea is that a Resource (identified by a URI) can be described through a collection of Statements forming a so-called RDF Description. A specific resource together with a named property and its value is an RDF statement. RDF/S

schemas are then used to declare vocabularies, i.e., collections of classes and properties, that can be used in resource descriptions for a specific application or domain.

RDF describes the interrelationships among resources in terms of named properties and values. These named properties may be thought of as attributes of resources and in this sense correspond to traditional attribute-value pairs. One main difference is that properties defined in RDF are only identified by their name (URI), not like in other object models, where attributes are identified by their name plus the domain class they can describe. So, properties have a URI and therefore are also resources. The value of a property can be another resource or a literal. A literal is simple string or another primitive data type.

We can see RDF through three different points of view - representations: (i) as RDF graph (ii) as RDF 3-tuples (the set of statements described in triples) (iii) RDF syntax (provides some standard ways for describing data using XML).

The RDF graph is a syntax-neutral way of representing RDF expressions using directed labeled graphs. These graphs are also called nodes and arcs diagrams. Each arc represents a named property. Each property connects two nodes, coming from a node representing a resource (drawn as oval) and pointing to another resource or a literal (drawn as rectangle). An example of the RDF graph is shown in Figure 2.3.

This RDF data model differs from the XML one in that it is a *labeled graph instead of a labeled tree*. RDF graph has labeled nodes and labeled arcs, while XML tree has only labeled nodes. Additionally, XML nodes are ordered, while in RDF graph there is no ordering. Finally, RDF data model suggests some more features, not available in other graph models, such as sets and bags.

The other way to represent RDF is 3-tuples, also called triples. Each triple {`s, p, o`} corresponds to an arc from the subject `s` to the object `o`, labeled by the predicate `p`. The following example shows that the arc named `sculpts` that connects the resource "`Michelangelo`" with the artifact "`Descent`" (in Figure 2.3) is written as:

```
{ <www.culture.net//michelangelo>,
  <www.icom.com/schema.rdf#Sculpts>,
  <http://www.photojournal.com/.../michelangelo/sculptur/descent.jpg>
}
```

This 3-tuple representation is an easy way to reenact a very huge set of statements, since it needs less space in comparison with the other two representations. However, it is not easily maintainable and lacks the XML's widely-deployed base of software and APIs, available for the third representation (the one that will be discussed soon).

The last representation of RDF (the one of XML syntax) is shown (for part of the graph representation of Figure 2.3) below:

```
<?xml version="1.0"  encoding="ISO-8859-1"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">

    <rdfs:Class rdf:ID="Artist"/>
    <rdfs:Class rdf:ID="Artifact"/>

    <rdf:Property rdf:ID="fname">
        <rdfs:domain rdf:resource="#Artist"/>
        <rdfs:range rdf:resource=".../XMLSchema-datatypes#string"/>
    </rdf:Property>

    <rdf:Property rdf:ID="lname">
        <rdfs:domain rdf:resource="#Artist"/>
        <rdfs:range rdf:resource=".../XMLSchema-datatypes#string"/>
    </rdf:Property>

    <rdf:Property rdf:ID="creates">
        <rdfs:domain rdf:resource="#Artist"/>
        <rdfs:range rdf:resource="#Artifact"/>
    </rdf:Property>

    <rdfs:Class rdf:ID="Sculptor">
        <rdfs:subClassOf rdf:resource="#Artist"/>
    </rdfs:Class>

    <rdf:Property rdf:ID="sculpts">
        <rdfs:subPropertyOf rdf:resource="#creates"/>
        <rdfs:domain rdf:resource="#Sculptor"/>
        <rdfs:range rdf:resource="#Sculpture"/>
    </rdf:Property>

    <Sculptor rdf:ID="michelangelo_id">
        <fname>Michelangelo</fname>
```

```
        <lname>Buonarroti</lname>
        <sculpts rdf:resource="#descent_id"/>
    </Sculptor>

    <Sculpture rdf:ID="descent_id">
        <title>Descent</title>
    </Sculpture>
</rdf:RDF>
```

2.2.1 RDF Query Language (RQL)

Since Semantic Web applications require the management of voluminous resource metadata, there is a need for a sufficiently expressive declarative language for querying both RDF descriptions and schemas. One of the proposed RDF query languages is called RQL [KAC+02]. It is a typed functional language and relies on a formal model for directed labeled graphs permitting the interpretation of superimposed resource descriptions by means of one or more RDF schemas. RQL adapts the functionality of semistructured/XML query languages to the peculiarities of RDF but, foremost, it enables to uniformly query both resource descriptions and schemas.

The innovation of RQL is that it can query at both schema and instance level, exploring the subclass-subproperty hierarchies and the multiple classification of the resources in a transparent (to the user) way. Another advantage is the ability to support generalized path expressions with variables in the classes names and properties names. Finally, it supports queries that use set operators, XML Schema data types, aggregation functions and operators for handling values.

The simplified BNF, shown in Table 2.4, gives the basic format of the RQL queries. These queries can be in the general form `Select...From...Where` but can also be in the format shown in most cases of rule 1. These queries usually retrieve schema information or apply aggregation functions (min, max, etc.).

The following RQL is a schema query and returns the subclasses of the resource called `Artist`,

```
subclassof(Artist)
```

while the following returns the titles of exhibited resources that have been created by a `Sculptor`

[1]	query	::=	*"subclassof"* *"("* query *")"*
			| *"superclassof"* *"("* query *")"*
			| *"superpropertyof"* *"("* query *")"*
			| *"subpropertyof"* *"("* query *")"*
			| *"topclass"*
			| *"topproperty"*
			| *"leafclass"* (*"("* query *")"*)?
			| *"leafproperty"* (*"("* query *")"*)?
			| ‘*"domain"* *"("* query *")"*
			| *"range"* *"("* query *")"*
			| *"typeof"* *"("* query *")"*
			| *"namespace"* *"("* query *")"*
			| *"count"* *"("* query *")"*
			| *"avg"* *"("* query *")"*
			| *"min"* *"("* query *")"*
			| *"max"* *"("* query *")"*
			| *"sum"* *"("* query *")"*
			| *"bag("* query, query? *")"*
			| *"seq("* query, query? *")"*
			| query set_op query
			| query comp_op query
			| query bool_op query
			| *"not"* query
			| sfw_query
			| *"exists"* var *"in"* query *"such that"* query
			| *"forall"* var *"in"* query *"such that"* query
[2]	sfw_query	::=	*"select"* (*"distinct"*)? projslist
			"from" rangeslist (*"where"* query)?
[3]	set_op	::=	*"union"* | *"intersect"*| *"minus"*
[4]	bool_op	::=	*"and"* | *"or"*
[5]	comp_op	::=	*"<"* | *">"* | *"="* | *"<="* | *">="* | *"!="* | *"like"*
[6]	projslist	::=	*"*"* | query (*","* query)*
[7]	rangeslist	::=	pathexpr (*","* pathexpr)*
[8]	pathexpr	::=	pathlem (*"."* pathlem)*
[9]	pathlem	::=	(*"{"* from_to *"}"*)? query (*"{"* from_to *"}"*)?
[10]	from_to	::=	(data_var)? (*";"* (class_var | identifier))? | class_var
[11]	var	::=	data_var | class_var | property_var
[12]	data_var	::=	identifier
[13]	class_var	::=	*"$"* identifier
[14]	property_var	::=	*"@"* identifier

Table 2.4: Simplified BNF Grammar for RQL

```
select Z, W
from Sculptor.creates{Z}.exhibited, {V}title{W}
where Z = V
```

2.2.2 A View Definition Language for RDF (RVL)

In the context of the Semantic Web, information consumers of the same information set, pose different requirements to the way they view information, hence necessitating the existence of mechanisms, such as views, to provide tailored access to the information sources. In parallel, the ability of a query language to support mechanisms for defining views over a data set has been recognized as an added-value functionality to the expressive power of a query language in traditional database systems, such as relational and object-oriented systems, but also in modern database applications based on the semistructured and XML data models.

Recognizing the need, a view definition mechanism for the Semantic Web has been proposed and it is called RVL [Mag03] [MTCP03]. Based on the data model of the Resource Description Framework (RDF/S) and by taking advantage of the expressiveness of the RQL query language for RDF/S graphs, this view definition language incorporates the functionality needed and the peculiarities of the underlying data model, in a uniform way. Being the first integrated effort for a view definition language specification, RVL exploits the RQL type system and the abstraction levels of an RDF/S graph to specify two operators, which are able to support all the necessary functionality. This minimality constitutes the most important advantage of RVL.

Figure 2.4 represents the construction of a virtual RVL view, given the source schemas.

Generally the definition of an RVL view is in 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
```

This definition has the following explanation:

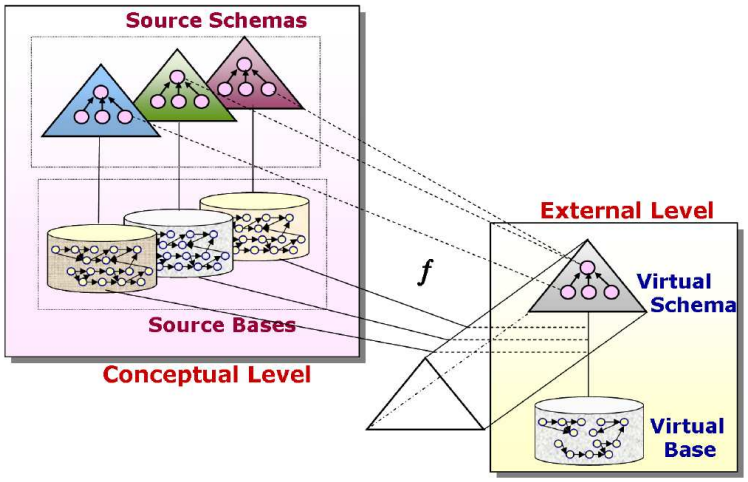


Figure 2.4: RVL virtual schemata

- The term `VIEW` used with the `operator` creates constructions of the type defined by the `operator`.
- The `FROM` clause is used for the evaluation of the variables defined in `VIEW` clause. `RQL_path_expression` cannot refer to part of the defined `VIEW`.
- `WHERE` clause contains the filtering conditions.
- `USING NAMESPACE` is used to define the prefix of a namespace that is used in the RQL query.
- `CREATE NAMESPACE` defines the URI of the namespace that is created in the view.

Given the definition above, we can see that the functionality of RVL exceeds the one of RQL. RQL is able to query RDF schemas and data, while RVL has the ability to construct virtual schemas and to instantiate them with the use of RQL variable bindings.

An example that could make more clear the above definition is:

```
VIEW rdfs:Class(PicassoWorksInOil);
```

This RVL statement "creates" the virtual class `PicassoWorksInOil`, while the

RVL statement below "populates" the virtual class with appropriate instances copied from the source schema defined in namespace `ns1`.

```
VIEW PicassoWorksInOil(Y)
FROM ns1:Artist{X}.paints{Y}.technique{Z}, {X}lname{W}
WHERE W like ‘‘*Picasso*’’ and Z like ‘‘oil*’’
USING NAMESPACE ns1=&http://www.icom.com/schema.rdf#
```

2.3 Comparison of XML and RDF

As discussed throughout this chapter the XML and the RDF data models are quite different. XML relies on a labeled tree model with labels on the nodes, which furthermore are ordered. On the other hand, RDF relies on a labeled graph model with labels on both nodes and edges. In this graph data model there is no ordering between the nodes. Sometimes this feature (ordering) consists a limitation of XML since, in many applications, the order of the information is restrictive.

In the XML data model, tree representation of the parent-child (and in general the ancestor-descendant) relationship yields the nesting of the tags in an XML document. In the RDF data model there is no such (parent-child) relation, and the nodes are linked with the help of properties. These properties may be considered to capture the nesting of XML nodes.

The first grammar proposed for XML documents, XML DTD, poses many restrictions to what can be expressed with it as discussed in Section 2.1.1. The fact that DTDs use only one atomic type is one limitation that we managed to get over with the use of XML Schema. RDF/S offers a large set of atomic types which leads to more refined RDF descriptions.

Additionally, DTDs do not support subsumption relationship. This means that in a DTD grammar we cannot specify that a `Vessel` is a subclass of a `Ceramic Object`. XML Schema incorporates the notion of type-derivation (types can be derived from one basetype), but this can only be partially compared to inheritance in RDF [KFvHH01]. RDF is able to specify single or multiple subsumption relationships. For example, we can specify that RDF/S class `Vessel` subsumes both `Ceramic Objects` and `Plastic Art Objects`.

Finally, DTDs and XML Schemas do not constrain the type of IDREFs. This indicates that we cannot specify that the attribute `exhibits`, defined as IDREF, refers to a `Museum`. This can be overcome by using keyref constraints of XML Schema. RDF, on the other hand, encounters this problem by specifying the domain and range of every property.

2.4 Conclusion

From the discussion in the previous sections, we can conclude that XML has some desirable characteristics. The first benefit of XML is that users are not restricted to a limited set of tags as in HTML. This enhances the readability of the XML document. Additionally, in XML there are many levels of structure. Starting from even not well-formed documents we may end up to valid documents with a DTD or with a more strict grammar given by an XML Schema. Furthermore, the use of namespaces advances the reusability and the flexibility of the XML language. The above characteristics render XML as one of the most popular languages. It, already, has a large base of users and supporting tools. Many of the new browser versions are able to understand XML, while it has gained the interest and the support of industry.

However, XML is not enough. The use of XML is feasible for collaboration in small communities. On the other hand, RDF provides a very simple yet powerful model for describing the information we use. It does this in a way that allows the computers that process this information to understand its meaning. Describing metadata for the web sources is of significant importance.

Although RDF has become popular, we need XML sources (native or virtual) in order to exploit the supporting tools for this language. So, there is a need for expressive languages that map one language to the other. This mapping should be a (semi)automatic process, since finding semantic mappings by hand is not efficient and decreases the maintainability and reliability of the system.

Chapter 3

Data Integration Systems

With the popularity of the Internet, access to data, independent of its physical storage location, has become highly facilitated. Additionally, users can access a variety of data sources that are related in some way, and combine the returned data in order to discover useful information, not physically stored into a single source.

Definition 3.1 *A data integration system, is a system that provides users with transparent access to a collection of related data sources as if these sources, as a whole, constitute a single data source. [CKT01]* □

The main objective of a data integration system is to facilitate users to focus on specifying *what* data they want, rather than on describing *how* to obtain them. To achieve this, the system provides an integrated view of the data stored in the underlying data sources. In a data integration system, users are interested mainly in querying the integrated data rather than updating the data through the integrated view.

There are some major issues that should be considered in a data integration system related to its design, modeling and operation. Even if these issues concern every data integration system, they are differentiated according to the variety of data sources that they integrate. The most important issues concern autonomy, queries and heterogeneity of the sources.

Autonomy of the data sources regards the ability of the sources to choose their own design, their own schema (if any), their own data model and their own management of

data. Usually data sources are created in advance of the integrated system and they can, autonomously, take decisions about their data. The data integration system is not, usually, informed for such changes a priori. Furthermore, data sources are able to choose whether and when they communicate with other components (communication autonomy). These kinds of autonomy are present in every integration system but they are differentiated. Issues, for example, concerning differences in choosing data model does not influence systems that integrate relational databases, since all the sources have a common data model.

Additionally, a significant issue is that of querying the integration system. Users pose queries formulated in terms of the provided integrated view and these queries should be translated into forms that are appropriate for the data sources. If the integrated sources are relational databases then user queries should be translated into a query language that is common for all the sources (i.e., SQL). However, when the data sources have different data formats this translation is not trivial. Furthermore, possible access limitations of the data sources for answering queries should be taken into account because some sources may be able to answer only a small fragment of queries. Additionally, traditional query optimization techniques, that usually depend on statistical models, may not be applicable, since such statistical models are not (easily) available for all the sources.

Finally, the problem of the discrepancies between the data sources is highly important. Usually, the contents of data sources are related in some way, but they show diversity in many aspects. This diversity, which is usually referred as *heterogeneity*, causes the design of a data integration system to be a challenging problem. Resolving the differences between the data sources is a crucial issue. There are different layers of heterogeneity beginning from *hardware* heterogeneities and continuing to discrepancies in the *operating systems* or *communication protocols*. On a higher level there is *logical* heterogeneity, which is the most intricate problem. Heterogeneity exists in any integration system, but in the case of older integration systems (relational database sources) some kinds of discrepancies do not exist. For example, there are no data model conflicts and every source has a schema, which is not ensured in nowadays

integrations systems. Heterogeneity is further discussed in Section 5.6.

3.1 Data Integration Architectures

Data integration systems can be classified according to their approach for managing sources. One of the possible classifications may be the one based on whether the queries to the integration system are sent directly to the data sources or whether there are results of the queries that are pre-stored. The *virtual view* approach corresponds to the former technique, while the *materialized view* approach uses pre-stored results.

3.1.1 Virtual View Approach

In the virtual view approach, the data are accessed from the sources on-demand when a user submits a query to the data integration system. The two representative architectures of this approach are federated database systems, and mediated systems. Despite of the fact that mediated systems have many similarities with the federated databases, there are some basic differences:

- In mediated systems data sources are not necessarily databases.
- Sources in a mediated system can be added or removed easily.
- Usually, unlike the FDBSs (where access is read/write), in a mediated system access in the sources is read only [CKT01]. This is due to the fact that sources in mediator-based systems are more autonomous.

3.1.1.1 Federated Database Systems (Read/Write Systems)

A federated database system (FDBS) consists of some semi-autonomous components that participate in federation to partially share data with each other. Each federated source can also operate independently from the others. These components are not fully autonomous in the sense that they are modified by adding an interface

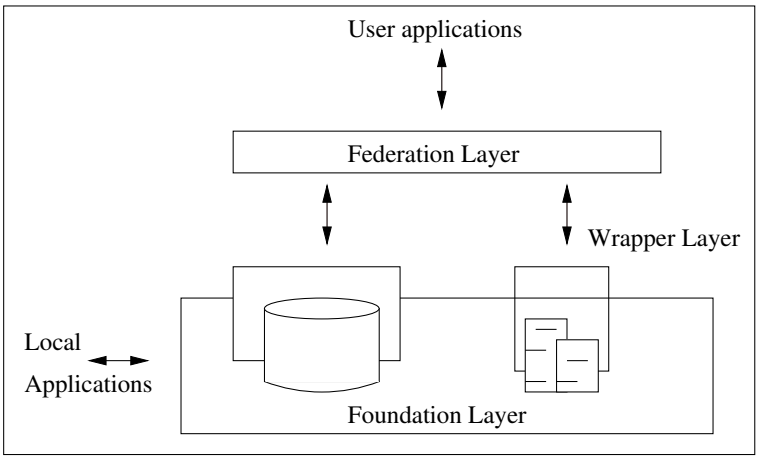


Figure 3.1: The architecture of a federated database system

that allows communication with all other databases in the federation. Each component of the federation is either a centralized DBMS, a distributed DBMS or another federated database management system. There are two basic types of FDBS: *tightly coupled* FDBSs and *loosely coupled* FDBSs.

A tightly coupled FDBS has *one or more unified schemas* which can be produced automatically or manually (by a user). In this type of FDBS, domain experts should undertake the arduous task of integrating all schemas of the federation into a global one. These FDBSs are static and it is very difficult to add or remove components from the integration system.

A loosely coupled FDBS does not have a unified schema, but it provides some *unified language for querying sources*. In this approach database components are more autonomous and they can decide how they will view all the accessible data in the federation. As there is no global schema, each source can create its own "federated schema" for its needs. Like the tightly coupled approach, logical heterogeneity should be resolved by domain experts.

The architecture of a federated database system is depicted in Figure 3.1. Federated databases is an approach appropriate to use when there is a small number of autonomous sources, and we want to retain their "independence", allowing user to query them separately and let them collaborate with each other to answer a query.

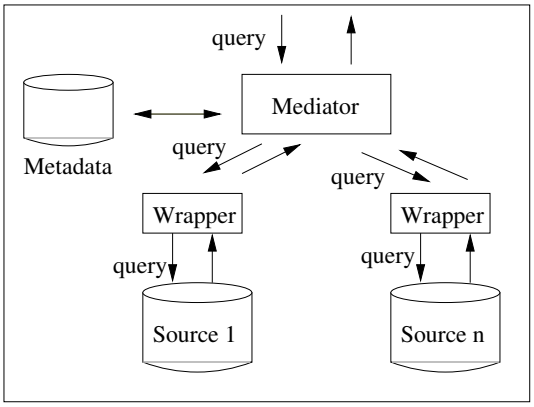


Figure 3.2: The architecture of a mediated integration system

3.1.1.2 Mediated Systems (Read Only Systems)

Mediated systems are an alternative architecture of data integration systems. They integrate data sources by providing a global virtual view. This global view is called *mediated schema*, and it is employed by the users in order to formulate their queries. The architecture of a mediated system is shown in Figure 3.2.

There are two basic software components of a mediated system: the *mediator* and one *wrapper* per data source. The former offers a common interface to a set of autonomous, independent and possibly heterogeneous data sources. The mediator (a.k.a *integrator*) performs the following actions: Firstly it receives a query formulated in terms of the unified schema and decomposes this queries into sub-queries. These queries are addressed to specific data sources. This decomposition is based on source descriptions, which play an important role in sub-queries' execution plan optimization. Finally, the sub-queries are sent to the wrappers of the individual sources, which transform them into queries over the sources. The results of these sub-queries are sent back to the mediator. At this point the answers are merged into one and returned to the user. Besides the possibility of asking queries, the mediator has no control over the individual sources.

The latter component (wrapper) is responsible for wrapping a data source in such a way that the source can interact with the rest of the integration system. It provides the mediator with data from the source that it is in charge of, as requested by the

query execution engine. In consequence, it presents a data source as a convenient database, with the right schema and data, appropriate for being understood and used by the mediator. This presentation schema may be different from the real one, i.e., the internal to the data source. A wrapper hides low-level (protocol) and data model details of the data source from the mediator. It is an important component of both a mediator based architecture and a data warehouse (see Section 3.1.2).

A key element in the mediator architecture is the set of *source descriptions*, i.e., the descriptions of the available sources and their contents, which is achieved by establishing the relationships (*mappings*) between the global schema and the local schemata. These descriptions can be represented by a set of logical formulas, similar to the way in which views are defined in terms of base tables in the relational data model. The language usually chosen for expressing these mappings is Datalog. There are several fragments of Datalog that are used, based on the existence of recursion, negation and arithmetic comparisons. The most common framework is the one of conjunctive queries (Datalog with relational predicates) with neither recursion nor negation, and arithmetic comparisons limited to equalities. There are different approaches with respect to how mappings are defined, and will be discussed thoroughly in Section 3.2.

3.1.2 Materialized View Approach

In the materialized view approach (a.k.a data warehousing) some filtered information from data sources is pre-stored (materialized) in a repository and can be queried later. The single repository in which data are stored is called *data warehouse*.

There are some important issues that should be taken into account for designing and maintaining a data warehouse. Firstly (designing phase) we need to decide what information from each source is going to be used, what views over these sources is going to be materialized and what global schema will be employed by the warehouse. Next (maintenance phase) we have to deal with how the warehouse is initially populated by the source data and how it is refreshed when the data in the sources are updated. Finally, there are some query processing, storage and indexing issues

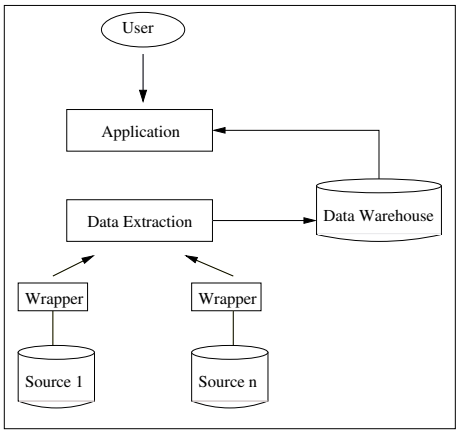


Figure 3.3: The architecture of a data warehouse

that should be taken into consideration. The architecture of the materialized view approach is depicted in Figure 3.3, where the Data Extraction component is responsible for the maintenance of the information stored in the Data Warehouse. This information is available to the user for querying and/or updating.

3.2 Modelling the Data Sources

Data integration systems can be classified according to *the way* in which the sources are described, i.e., how their content is related to the mediator global schema. There are two main approaches: the Global-As-View approach (GAV) and the Local-As-View approach (LAV). Furthermore, hybrid approaches based on both GAV and LAV have been recently proposed.

Definition 3.2 *A data integration system is a triple* $\mathcal{I}\langle \mathcal{G}, \mathcal{S}, \mathcal{M}_{\mathcal{G},\mathcal{S}} \rangle$ *[LLR02] where*

- $\mathcal{G}$ *is the global schema expressed in the global language* $\mathcal{L}_{\mathcal{G}}$ *over alphabet* $\mathcal{A}^{\mathcal{G}}$. *The language* $\mathcal{L}_{\mathcal{G}}$ *determines the expressiveness allowed for specifying the global schema, i.e., the set of constraints that can be defined over it.*

- $\mathcal{S}$ *is the set of the local schemas. It is modeled in the source language* $\mathcal{L}_{\mathcal{S}}$ *over the alphabet* $\mathcal{A}^{\mathcal{S}}$. *As in the case of global schema the language determines the set of constraints that can be defined over it.*

- $\mathcal{M}_{\mathcal{G},\mathcal{S}}$ *is the mapping between* $\mathcal{G}$ *and* $\mathcal{S}$.□

In order to specify the semantics of a data integration system, we start with a set of data at the sources and specify which data satisfies the global schema. We start with a source database for $\mathcal{I}$, the database $\mathcal{D}$ for the source schema $\mathcal{S}$. Based on this $\mathcal{D}$, we have to specify which information content of the global schema $\mathcal{G}$ is. Any database for $\mathcal{G}$ is called a global database for $\mathcal{I}$. A global database $\mathcal{B}$ for $\mathcal{I}$ is said to be *legal* with respect to $\mathcal{D}$, if:

- $\mathcal{B}$ is coherent with $\mathcal{G}$, i.e., every constraint in schema $\mathcal{G}$ is satisfied by $\mathcal{B}$.
- $\mathcal{B}$ satisfies the mapping with respect to $\mathcal{D}$, that is its tuples respect the relationships defined between the global and the source schema.

Definition 3.3 *Given a source database* $\mathcal{D}$ *for* $\mathcal{I}$*, the semantics of* $\mathcal{I}$ *with respect to* $\mathcal{D}$*, denoted* $sem(\mathcal{I}, \mathcal{D})$*, is defined as:*

$$sem(I, D) \;=\; \{\; B \mid B \textit{ is a legal global database for } I \textit{ w.r.t. } D \;\}$$

□

Let us now turn our attention to queries. In order to define the semantics of a query q over the global schema $\mathcal{G}$, we have to take into account all the legal global databases for $\mathcal{I}$ with respect to $\mathcal{D}$. We call *certain answers* of a query q of arity n with respect to $\mathcal{I}$ and $\mathcal{D}$, the set $q^{\mathcal{I},\mathcal{D}}$ of n-tuples t such that $t \in q^{\mathcal{D}\mathcal{B}}$ for every database $\mathcal{D}\mathcal{B} \in sem(\mathcal{I}, \mathcal{D})$. Certain answers is what we call, answers to a user query.

Given the above formal definitions, the mapping $\mathcal{M}_{\mathcal{G},\mathcal{S}}$ between the global schema and the sources is provided in terms of a set of assertions of the form $\langle R,\ V\rangle$, where R is a view over the global schema $\mathcal{G}$ and V is the view over the source schema $\mathcal{S}$.

Associated to each mapping assertion $\langle R,\ V\rangle$ we have a specification *as(V)* of which *assumption* to adopt for the view V, i.e., given a source database $\mathcal{D}$, how to interpret $R^{\mathcal{D}}$ with respect to the set of tuples in the answer to V over a global database $\mathcal{B}$, i.e., $V^{\mathcal{B}}$.

Definition 3.4 *The assumption we adopt for a view V, called $as(V)$, to each mapping assertion $\langle R,\ V\rangle$ is defined as follows:*

When $as(V)$ = sound, the extension of the associated global view R provides any superset of the tuples satisfying V. In other words, from the fact that a tuple is in $V^{\mathcal{D}}$ one can conclude that it satisfies the corresponding global relation R, while from the fact that a tuple is not in $V^{\mathcal{D}}$ one cannot conclude that it does not satisfy R. Formally a global database $\mathcal{B}$ satisfies the sound view V if $V^{\mathcal{D}} \subseteq R^{\mathcal{B}}$

When $as(V)$ = complete, the extension of the associated global view R provides any subset of the tuples satisfying V. In other words, from the fact that a tuple is in $V^{\mathcal{D}}$ one cannot conclude that such a tuple satisfies R. On the other hand, from the fact that a tuple is not in $V^{\mathcal{D}}$ one can conclude that such a tuple does not satisfy R. Formally, a global database $\mathcal{B}$ satisfies the complete view V , if $V^{\mathcal{D}} \supseteq R^{\mathcal{B}}$.

When $as(V)$ = exact, the extension of the associated global relation R is exactly the set of tuples satisfying V . Formally, a global database $\mathcal{B}$ satisfies the exact view V, if $V^{\mathcal{D}} = R^{\mathcal{B}}$. □

3.2.1 Global Centric Approach

In the Global-As-View approach (GAV) [Ull00], the global schema is defined in terms of the local sources' schemas. That is, the global schema is defined as a *view* over the local sources' schemas. An overview of the approach is illustrated in Figure 3.4, where concept G_1 of the global schema is expressed in terms of the relations S_1 and S_2 of the local database sources. If both these schemas are relational, then one can write a *rule-based conjunctive query* over the source relations. This query specifies how to obtain the tuples for the global schema relations. Each query specifies that in order to compute the tuples in the relation in the *head* of the rule, one has to go to the *body* of the rule and compute whatever is specified there. The attributes appearing in the head indicate that they are the attributes of interest, thus the others (in the body) can be projected out at the end. If there is more than one rule to compute the same relation, we use all of them and we take the union of the results.

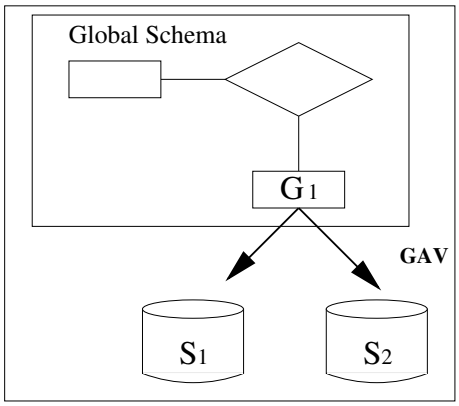


Figure 3.4: Global centric approach

Definition 3.5 *Given the Definition 3.2 mappings ($\mathcal{M}_{\mathcal{G},\mathcal{S}}$), in GAV approach, are in the form:*

$$\mathcal{G}_i(\overline{X}) \leftarrow \mathcal{S}_1(\overline{X_1},\ \overline{Y_1}),\ \mathcal{S}_2(\overline{X_2},\ \overline{Y_2}),\ ...,\ \mathcal{S}_k(\overline{X_k},\ \overline{Y_k})$$

where $\overline{X} = \bigcup_i \overline{X_i}$ and $\mathcal{G}_i$, $\mathcal{S}_i$ are global and local relations respectively.□

Example 3.1 *Supposing that we are interested in the cultural domain and both local and global schemas are expressed as relations. Our example global schema consists of three relations:*

***Artists(Name, Year of Birth)**,*

where the two attributes represent the name and the year of birth of each particular artist.

***Artifact(Title, Style, Artist, Kind)**,*

where Title is the name of the artifact, Style refers to the technique used for the creation of the artifact, Artist refers to the creator of the artifact and Kind is the type of the artifact (e.g., sculpture, painting).

***ExhibitedInMuseum(Title, Museum, Period)**,*

where the Title represents the name of the artifact, Museum is the name of the museum that the artifact is exhibited in, and Period is the time period that the artifact was exhibited in this particular museum.

Now assume that there are three local sources which consist of one relation S1, S2 and S3 respectively. S1 contains the name of the artist and the year of their birth. S2 contains the id of an artifact, its title, kind, style and the artist that created it. Finally, S3 contains information about the museums, the artifacts exhibited in these museums and the period of the exhibition.

The queries that specify how to obtain the tuples for the global schema relations are the following:

Artist(Name, Year of Birth)	$\leftarrow$	*S1(name, year of birth)*
Artifact(Title, Style, Artist, Kind)	$\leftarrow$	*S2(id, title, style, kind, artist)*
ExhibitedInMuseum(Title, Museum, Period)	$\leftarrow$	*S3(museum, id ,period),*
		S2(id, title, style, kind, artist)

These queries state that tuples for the global schema relation Artist are constituted by source S1, Artifact tuples are constituted by source S2 and ExhibitedInMuseum tuples are constituted by a join of the tuples originated from sources S2 and S3[1].□

Definition 3.6 *Given the formal definitions 3.4, 3.5, the mapping $\mathcal{M}_{\mathcal{G},\mathcal{S}}$ between the global schema and the sources, constitutes of the following assertions:*

$r^{\mathcal{B}} \supseteq V^{\mathcal{D}}$ *(sound source)*

or

$r^{\mathcal{B}} \equiv V^{\mathcal{D}}$ *(exact source)*

where r is a relation of the global schema, and V is a view (query) over the local sources. □

It is an implicit assumption [Len02], in many GAV proposals, that the assertions above are exact. This assumption is true when in the global schema there are *no additional* constraints. Under these circumstances, the query rewriting in GAV approach is quite easy (this will be illustrated below). However, the possibility of specifying constraints in the global schema enhances the expressing power of GAV systems.

[1]Global schema attributes' names start with a capital letter and the corresponding attributes from the local schemas start with a lowercase letter.

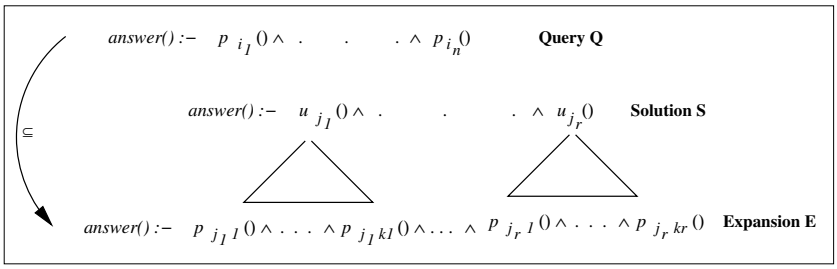


Figure 3.5: Query unfolding

As depicted in Figure 3.5 [Ull00] the Global-As-View approach has the advantage of simple query rewriting. Due to the fact that global schema is expressed in terms of the local schemas query rewriting consists of replacing each atom of the query with its definition. In this way the query is finally expressed in terms of the local sources. This substitution is called *query unfolding*.

Example 3.2 *Based on Example 3.1 we are interested in a query (written as rule-based conjunctive query) returning the title, the style and the period of paintings, created by Da Vinci and exhibited in the Louvre:*

$$\begin{aligned} \textit{query(Title, Style, Period)} \quad \leftarrow \quad & \textit{ExhibitedInMuseum(Title, 'Louvre', Period),} \\ & \textit{Artifact(Title, Style, 'Da Vinci', 'Painting')} \end{aligned}$$

With the help of the descriptions for the global schema, the resulting query will be:

$$\begin{aligned} \textit{query'(Title, Style, Period)} \quad \leftarrow \quad & \textit{S3('Louvre', id, Period),} \\ & \textit{S2(id, Title, Style, 'Painting', 'Da Vinci')} \end{aligned}$$

In query' relations ExhibitedInMuseum and Artifact have been replaced by S2, S3 and S2 respectively, but only one appearance of S2 is kept.□

The major drawback of this approach is its lack of flexibility with respect to the addition/deletion of the sources to the data integration system, or the modification of the sources schemas. This is due to the fact that each modification of a local source schema results in modification of global schema.

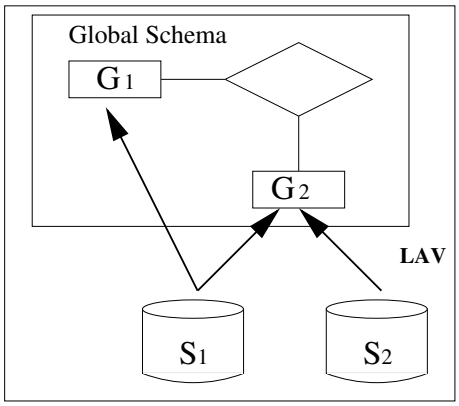


Figure 3.6: Local centric approach

3.2.2 Local Centric Approach

In the Local-As-View approach (LAV) [Lev99] the global schema is defined *independently* of the local sources schemas. Each source is described in terms of the global schema relations. That is, the sources are described as *materialized view* of the global schema. An overview of the LAV approach can be seen in Figure 3.6 where S_1 can be seen as a view over concepts G_1 and G_2, while S_2 can be seen as a view over G_2.

Definition 3.7 *Given the Definition 3.2 mappings ($\mathcal{M}_{\mathcal{G},\mathcal{S}}$), in LAV approach, are in the form:*

$$\mathcal{S}_i(\overline{X}) \leftarrow \mathcal{G}_1(\overline{X_1},\ \overline{Z_1}),\ \mathcal{G}_2(\overline{X_2},\ \overline{Z_2}),\ ...,\ \mathcal{G}_j(\overline{X_j},\ \overline{Z_j})$$

where $\overline{X} = \bigcup_i \overline{X_i}$ and $\mathcal{G}_i$, $\mathcal{S}_i$ are global and local relations respectively. □

Example 3.3 *Let's assume again that the global schema consists of the three relations defined in Example 3.1. Suppose that there are two sources S1 and S2 that are integrated in this data integration system. The descriptions of these sources are:*

S1(Name, Title)	$\leftarrow$	*Artifact(Title, Style, Artist, Kind),*
		Artist(Name, Year of Birth),
		Name = Artist, Year of Birth > 1900
S2(Title, Period)	$\leftarrow$	*Artifact(Title, Style, Artist, Kind),*
		ExhibitedInMuseum(Title, Museum, Period),
		Kind = 'Painting', Museum = 'Louvre'

From the descriptions above we can conclude that relation S1 contains the title of artifacts and the name of their creator, only for creators born after 1900, while source S2 contains the titles of paintings along with the period of their exhibition in the Louvre museum.□

Definition 3.8 *Given the formal definitions 3.4, 3.7, the mapping $\mathcal{M}_{\mathcal{G},\mathcal{S}}$ between the global schema and the sources, constitutes of the following assertions:*

$s^{\mathcal{D}} \subseteq V^{\mathcal{B}}$ *(sound source)*

or

$s^{\mathcal{D}} \equiv V^{\mathcal{B}}$ *(exact source)*

where s is a relation of the local schema, and V is a view (query) over the global schema relations. □

Due to the fact that the source relations are expressed as views over the global schema, each modification/addition/deletion of sources is costless since the local sources' schemas are the only things that should change. However, query rewriting in this approach is quite complex. The user of the data integration system poses a query in terms of the global schema and this query should be translated in terms of the local ones. The Local-As-View approach provides the system with a description of the local sources over the global schema and no description in the other "direction", as in the Global-As-View approach. So, it is not possible for the query to obtain the answers by a simple and direct computation of the body of the query. However, the data reside in the sources, and the query should be translated against them. This problem is widely known as the problem of *rewriting queries using views.*

The problem of query rewriting using views can become clearer if we consider the following: Let's assume that we have a collection of views $V_1, V_2, ..., V_n$, whose contents have already been computed, and cached or materialized. When a new query Q arrives, instead of computing its answers directly, we try to use the answers stored to $V_1, V_2, ..., V_n$. A problem to consider lies in determining *how much* from the real answer we get by using the pre-computed views only; and also determining *the maximum* we can get in terms of the kind of views we have available.

3.2.3 Query Reformulation Algorithms

Definition 3.9 *With respect to the Definition 3.2 we assume a query Q expressed over the alphabet $\mathcal{A}^{\mathcal{G}}$ and a rewriting Q' expressed over the alphabet of the sources $\mathcal{A}^{\mathcal{S}}$. Given this query Q and a set of views $V = \{V_1, \ldots, V_n\}$ over the A^G, the expression Q' defined over $\mathcal{A}^{\mathcal{S}}$ is a rewriting of Q if $Q' \cup V$ is contained in Q and Q' does not refer to the predicates of A^G.*

The problem of query containment for conjunctive queries is NP-complete [SY80]. Below, we study four algorithms that deal with the problem of query rewriting.

3.2.3.1 The Bucket Algorithm

The main idea underlying the Bucket Algorithm [LSK95] is that we can reduce the number of query rewritings that needs to be considered if we consider each subgoal in the query separately to determine which views may be relevant to each subgoal. Given a query Q the Bucket Algorithm finds a rewriting of Q in two steps:

1. The algorithm creates a bucket for each subgoal in Q that contains the views (i.e., data sources) that are relevant to answering that particular subgoal.

2. The algorithm tries to find query rewritings that are conjunctive queries, each consisting of one conjunct from every bucket. For each possible choice of element from each bucket, the algorithm checks whether the resulting conjunction is contained in the query Q, or whether it can be made to be contained if additional

predicates are added to the rewriting. If so, the rewriting is added to the answer. Hence, the result of the Bucket Algorithm is a union of conjunctive rewritings.

The algorithm above can become more clear with the following example:

Example 3.4 *Let's assume again that the global schema consists of the three relations defined in Example 3.1. Suppose that there are two sources S1 and S2 that are integrated in this data integration system. The descriptions of these sources are the same as in Example 3.3. Suppose that the query posed by the user to the global schema calculates the sculptures' title, along with their creator and creator's year of birth:*

$$\begin{aligned} Q(Title,\ Name,\ Year\ of\ Birth) \ \leftarrow\ & Artifact(Title,\ Style,\ Artist,\ Kind), \\ & Artist(Name,\ Year\ of\ Birth), \\ & Kind = \text{`Sculpture'},\ Artist = Name \end{aligned}$$

The buckets that are constructed for this example using the first step of the Bucket Algorithm are:

$Artifact(Title,\ Style,\ Artist,\ Kind)$	$Artist(Name,\ Year\ of\ Birth)$
$S1(Name,\ Title)$	$S1(Name,\ Title)$

The first line shows the subgoals of the user query whereas the second line shows the sources Si that are related to these subgoals. It is true that subgoal Artifact is also related with S2 but the restriction of Kind (Kind = 'Sculpture') in the query does not allow us to add the source S2 (Kind = 'Painting'), since they are contradictory.

The second step of the algorithm chooses one view from each bucket and it combines them into a new query. In this example we have already one entry per bucket, so there is only one combination of views. In general, we should construct one query per possible combination of entries and test for containment in the original query. Then, the result would be the union of all the contained queries.

The new query, written in terms of the sources is:

$$Q'(Title,\ Name,\ Year\ of\ Birth) \ \leftarrow\ S1(Name,\ Title)$$

In this query, the second appearance of S1 has been eliminated.□

This algorithm is able to reformulate queries of the form:

$$Q(\bar{X}) \ :- \ C(\bar{Y}),\ S_1(\bar{X}_1),\ \ldots,\ S_k(\bar{X}_k)$$

It concerns, conjunctive queries where $S_i(\bar{X}_i)$'s are first-order predicates and $C(\bar{Y})$ is a conjunction of *order constraints* on the variables of the query. Order constraints are of the form $\alpha_1 \theta \alpha_2$, where α_1 and α_2 are variables or constants, and $\theta \in \{<, \leq, =, \neq\}$.

It is guaranteed to find a maximally contained rewriting of the query if it does not contain arithmetic comparison predicates and there are no functional dependencies over the global schema. The Bucket algorithm has to perform a lot of containment tests and this is quite expensive, since testing containment of conjunctive queries is NP-complete [GMUW01].

3.2.3.2 Inverse Rules Algorithm

The key idea underlying this algorithm is to construct a set of rules inverting the view definitions, i.e., rules that show how to compute tuples for the mediated schema relations from tuples of the views [Dus97] [DGL00] [Lev01]. One can think of this process as obtaining GAV definitions out of LAV ones. One inverse rule is constructed for every subgoal in the body of the view. While inverting the view definitions, the existential variables that appear in the view definitions are mapped using Skolem functions to ensure that the value equivalences between the variables are not lost.

Example 3.5 *Let's consider a slightly different description of the source S1 given in Example 3.3:*

S1(Artist, Period)	←	*Artifact(Title, Style, Artist, Kind),*
		ExhibitedInMuseum(Title, Museum, Period)

For every subgoal of the view definition we should write an inverse rule:

Artifact(f_1(Artist, Period), f_2(Artist, Period), Artist, f_3(Artist, Period))
← S1(Artist, Period)

ExhibitedInMuseum(f_1(Artist, Period), f_4(Artist, Period), Period)
← S1(Artist, Period)

In this algorithm, the attributes Title, Style, Kind and Museum are replaced by a Skolem function f_i which takes all the attributes of the view (S1) head as input. The rewriting of a query Q using the set of views V is the Datalog program that includes the inverse rules for V and the query Q. Given the query:

$$\begin{aligned} Q(Artist, Period) \leftarrow\ & Artifact(Title, Style, Artist, Kind), \\ & ExhibitedInMuseum(Title, Museum, Period) \end{aligned}$$

This query is reformulated to (where for simplicity we assume that T=Title, S=Style, A=Artist, K=Kind, M=Museum, and P=Period):

$$\begin{aligned} Q(A, P) \leftarrow\ & Artifact(f_1(A, P), f_2(A, P), A, f_3(A, P)), \\ & ExhibitedInMuseum(f_1(A, P), f_4(A, P), P), \\ & T=f_1(A, P), S=f_2(A, P), K=f_3(A, P), \\ & M=f_4(A, P) \end{aligned}$$

By using the inverse rules defined previously we conclude into:

$$\begin{aligned} Q(A, P) \leftarrow\ & S1(A, P), S1(A, P), \\ & T=f_1(A, P), S=f_2(A, P), K=f_3(A, P), \\ & M=f_4(A, P) \end{aligned}$$

By eliminating what we don't need and by replacing the variables introduced before, we come up with:

$$Q(Artist, Period) \leftarrow S1(Artist, Period)$$

□

This algorithm reformulates recursive conjunctive queries in presence of functional or full dependencies. It is guaranteed to find a maximally contained rewriting in polynomial time in the size (number of predicates) of the query and the views.

3.2.3.3 MiniCon Algorithm

MiniCon Algorithm [PL00] is an improved version of the Bucket Algorithm. As in the Bucket Algorithm, there are two steps: computing the buckets (one for each subgoal of the query) and then computing the rewritings by using the buckets. Additionally, MiniCon Algorithm is attentive to the interaction of the variables in the query and in the view definitions, in order to prune some of the views that will be added into the buckets. This way, the number of views to be considered for the rewriting step is reduced. MiniCon Algorithm considers conjunctive queries, as Bucket Algorithm does.

Example 3.6 *Let's consider a slightly different description of the sources given in Example 3.3:*

S1(Name, Style)	←	*Artifact(Title, Style, Artist, Kind),*
		Artist(Name, Year of Birth),
		Name = Artist, Year of Birth > 1900
S2(Title, Period)	←	*Artifact(Title, Style, Artist, Kind),*
		ExhibitedInMuseum(Title, Museum, Period),
		Kind = 'Painting', Museum = 'Louvre'

Suppose that the query posed by the user to the global schema calculates the title of the paintings and the period of their exhibition:

Q(Title, Period)	←	*Artifact(Title, Style, Artist, Kind),*
		ExhibitedInMuseum(Title, Museum, Period),
		Kind = 'Painting'

The buckets that are constructed for this example with the help of the first step of the Bucket Algorithm are:

Artifact(Title, Style, Artist, Kind)	*ExhibitedInMuseum(Title, Museum, Period)*
S2(Title, Period)	*S2(Title, Period)*
S1(Name, Style)	

The first line shows the subgoals of the user query whereas the other lines show the sources Si that are related to these subgoals.

By examining the variables of the source S1, it becomes obvious that it does not contribute to the query, since it does not contain the variable in which the subgoals of the query Q joins (Title). So, it can be removed from the bucket, leaving only source S2.

The new query, written in terms of the sources is:

$$Q'(Title, Period) \leftarrow S2(Title, Period)$$

In this query, the second appearance of S2 has been eliminated.□

The key advantage of the MiniCon Algorithm is that, in the end, it considers fewer combinations that the cartesian product of the views in the buckets created by the Bucket Algorithm. The complexity of the algorithm is exponential in terms of the number of the views, but in general it outperforms both the Bucket and the Inverse Rules Algorithm.

3.2.3.4 The Shared Variable Bucket Algorithm

This algorithm [Mit99], like the MiniCon, also aims at recovering the weak aspects of the Bucket Algorithm to achieve a more efficient query reformulation. It examines conjunctive queries, like the ones taken into consideration on the Bucket Algorithm. The idea is again to examine the shared variables and reduce the bucket contents so, that the number of view combinations to be considered is reduced in the second phase of the algorithm. The complexity of the algorithm is in the worst case exponential with respect to the size of the query and the views.

3.2.4 Combining Global and Local Approach

As discussed earlier, both GAV and LAV have some drawbacks that should be overcome. Thus, an approach has been proposed that combines global and local approach. It is called *GLAV* [FLM99]. In this approach we are able to express a

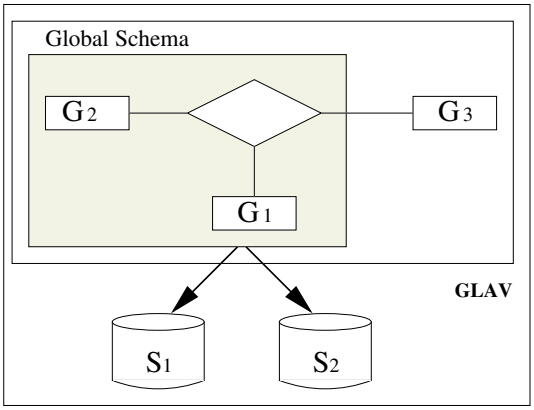


Figure 3.7: GLAV approach

local source in terms of the global schema (LAV), a global source in terms of the local sources (GAV) and *additionally* a whole view ("part") of the global schema in terms of the local sources. An overview of this approach can be seen in Figure 3.7 where the whole "relation" of G_1 and G_2 is described by the local sources relations, S_1 and S_2.

Definition 3.10 *Given the Definition 3.2 mappings ($\mathcal{M}_{\mathcal{G},\mathcal{S}}$), in GLAV approach, are in the form:*

$$\mathcal{G}_1(\overline{X_1},\ \overline{Z_1}),\ \mathcal{G}_2(\overline{X_2},\ \overline{Z_2}),\ ...,\ \mathcal{G}_j(\overline{X_j},\ \overline{Z_j}) \leftarrow \mathcal{V}(\overline{X},\ \overline{Y})$$

where $\overline{X} = \bigcup_i \overline{X_i}$, $(\bigcup_i \overline{Z_i}) \bigcap \overline{Y} = \emptyset$, $\mathcal{G}_i$ are global relations and $\mathcal{V}(\overline{X},\ \overline{Y})$ is a conjunction of source relations. □

Example 3.7 *Let's assume that the global schema consists of the three relations described in example 3.1 and that there are two local sources S1 and S2. S1 describes the artifacts, while S2 describes the artifacts that are exhibited in a museum for a certain period and the artist that created them. The queries that specify how to obtain the tuples for the global schema relations are the following:*

Artifact(Title, Style, Artist, Kind),	←	*S2(title,artist,museum,period)*
ExhibitedInMuseum(Title, Museum, Period)		
Artifact(Title, Style, Artist, Kind)	←	*S1(id,title,style,kind,artist)*

The second query describes the global relation Artifact in terms of the local source S1 (typical GAV approach), while in the first query the `conjunction` *of the two relations (Artifact and ExhibitedInMuseum) is described in terms of the source S2. This type of descriptions differentiates the GLAV approach.*□

Definition 3.11 *Given the formal definitions 3.4, 3.10, the mapping* $\mathcal{M}_{\mathcal{G},\mathcal{S}}$ *between the global schema and the sources, constitutes of the following assertions:*

$R^{\mathcal{B}} \supseteq V^{\mathcal{D}}$ *(sound source)*

or

$R^{\mathcal{B}} \subseteq V^{\mathcal{D}}$ *(complete source)*

or

$R^{\mathcal{B}} \equiv V^{\mathcal{D}}$ *(exact source)*

where R is a view (query) of the global schema, and V is a view (query) over the local sources.□

The GLAV approach combines the expressive power of GAV and LAV. Additionally it reaches the limits of the expressive power of a data source description language. This is because slight additions to the expressive power of GLAV would make query answering co-NP-hard in the size of the data in the sources. Query rewriting in this approach is shown to be no harder than it is for the LAV approach. It should be noted that GLAV is also of interest for data integration independent of data sources, because of the flexibility it provides in integrating heterogeneous sources.

3.2.4.1 Another Hybrid Approach: BAV

BAV is another data integration approach. BAV transformation sequences are partially derived from LAV or GAV view definitions. BAV is a rich integration framework, which is based on the use of reversible sequences of primitive schema transformations, called *transformation pathways*. It is an expressive data integration language, since it allows the expression of mappings in both directions. Another major advantage of using the BAV approach is that it supports the evolution of both global and local schemas, in contrast to taking either a GAV or LAV approach. With

the BAV approach it becomes possible to extract a definition of the global schema as a view over the local schemas and vice versa. BAV combines the benefits of LAV and GAV in the sense that any reasoning or processing which is possible with the view definitions of GAV or LAV will also be possible with the BAV definition. However, BAV is likely to be more costly to reason with and process than the corresponding LAV, GAV or GLAV view definitions would be. The most representative system that implements this approach is AutoMed [BLK+04] [JTMP04].

3.2.5 Comparison of the Approaches

In conclusion, there are some interesting facts that should be noted about the approaches above. It is true that both LAV and GAV, which are the first approaches proposed, have advantages and disadvantages. LAV is really flexible in addition/deletion of the local sources that participate in the integration system; this is the main drawback of the GAV approach, since every addition/deletion leads to a new rewriting of the global schema description. Exactly the same occurs when it is necessary to add some more complicated constraints on sources, since LAV demands only the addition of the necessary changes to the source, while GAV demands the rewriting of the global schema description. These constraints usually concern the availability of the data during querying the sources. However, query answering is quite simple in GAV, whereas it is harder in LAV. In the GAV approach, query rewriting can be achieved by unfolding the source descriptions of the global "parts" of the query. In the LAV approach, this is not feasible since such descriptions are not available. In the GLAV approach, the source evolution and addition/removal is easier, since, in fact, both directions are implemented (LAV and GAV) and in this case it is appropriate to use the LAV approach. Additionally, query answering is not harder than in LAV. However, GLAV (as will be discussed in Chapter 5) gives the ability of more expressive mappings.

An interesting observation, as stated in [CCGL02], is that, under certain circumstances, the existence of constraints (e.g., keys or foreign keys) in the global schema can turn the GLAV mappings into GAV ones, and thus take benefit of the query re-

formulation algorithm proposed for the GAV approach. Unfortunately, despite their expressiveness, GLAV mappings introduce new challenges. Recent work [MH03] has shown that the composition of GLAV mappings may be undecidable in certain cases (e.g., the composition of GLAV rules, which map non CQ_k[2] queries over a source schema to non CQ_k queries over a target schema, may result in an infinite set of mappings). Additionally, as illustrated in [FKPT04] the composition of two GLAV rules does not always imply a new GLAV rule that maps a conjunctive query to another conjunctive query. There are cases (e.g., when we compose two finite sets of non full[3] source-to-target dependencies used for the interpretation of the mappings) where the composition is definable only with the use of *existential second-order formulas*. In these formulas, new function symbols that guarantee the presence of the existentially quantified variables appearing in the dependencies, are introduced.

3.2.6 Representative Systems

Below we discuss briefly the most representative systems of the above source modeling approaches, namely Tsimmis, MIX, Nimble, Information Manifold, Infomaster, Agora, MARS, Dis@Dis.

Table 3.1 summarizes the basic characteristics of these information integration systems.

3.2.6.1 GAV Systems

One of the first systems (and the most representative of GAV approach) is Tsimmis system [CGMH+94]. This system uses the OEM (Object Exchange Model) [PMW96] to convey information between the components of the system. The first basic component of a mediated system, the mediator, is specified using MSL (Mediator Specification Language). It is a logic-based, object-oriented language that can be seen as a view definition language, targeted to the OEM data model. The second one is the

[2]Informally, CQ_k is the class of conjunctive queries in which every nested expression has at most k variables [MH03].

[3]A dependency is full if no existentially quantified variables occur in it.

	Approach	Sources	Global View	Query Language	Query Reformulation Algorithm
Tsimmis	GAV	Text files RDB OODB	OEM	LOREL/ MSL	Query Unfolding
MIX	GAV	XML	XML	XMAS	Query Unfolding
Nimble	GAV	XML	XML	XML-QL	Query Unfolding
Information Manifold	LAV	Structured files,OODB RDB, www form-based bases	Relational	SQL	Bucket Algorithm, MiniCon
Infomaster	LAV	Semi-structured	Relational	conjunctive queries	Inverse Rules Algorithm
Agora	LAV	RDB/XML	XML DTD	XQuery	Query rewriting using SQL views
MARS	GLAV	RDB/XML	XML	XQuery	Chase/ Backchase
Dis@Dis	GLAV	Relational	Relational	union of conjunctive queries	Variant of inverse rules
AutoMed	BAV	Relational XML and structured files	Relational	IQL	Query Unfolding

Table 3.1: Basic characteristics of the information integration systems

wrappers which are specified using WSL (Wrapper Specification Language). WSL is an extension of MSL, supporting the description of source contents and source query capabilities. End users pose queries, which are written in LOREL [AQM+96], an extension of OQL targeted to semi-structured data. LOREL queries are translated to MSL queries, which are forwarded to the mediator. The architecture of the Tsimmis is illustrated in Figure 3.8. An important assumption in the Tsimmis system, is that sources to be integrated have a common way to identify their objects.

Other GAV systems are: MIX and Nimble. MIX (stands for Mediation of Infor-

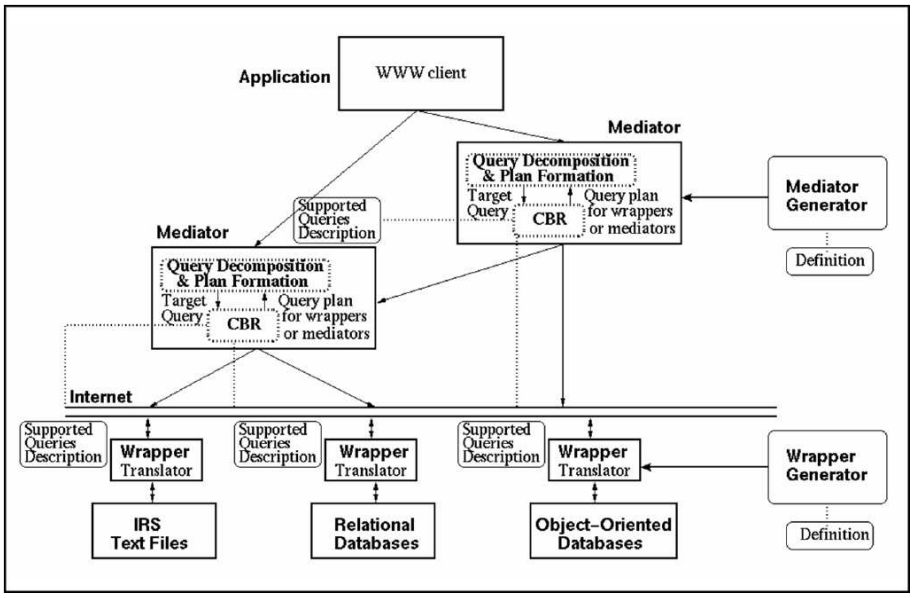


Figure 3.8: The basic components of Tsimmis System

mation using XML) [BGL+99] is a successor of Tsimmis. The basic difference from Tsimmis is that XML is used as the language (i) to represent the global schema and (ii) to exchange data between the mediator and the XML sources (instead of OEM). The query language of MIX is XMAS (XML Matching And Structure Language) instead of LOREL. XMAS uses features from several XML query languages and it allows object fusion and pattern matching on XML data. XMAS queries are formulated in terms of the mediator schema, and are rewritten into XMAS queries that refer to the source views exported by the wrappers. These queries are then sent to the wrapped sources for evaluation.

Similar to the MIX system is the commercial system called Nimble [DLW01]. Nimble integrates XML sources too. The architecture of Nimble system is based on a set of mediated schemas, which are defined as views over the schemas of the data sources. Similar to Tsimmis, the mediated schemas can be built in a hierarchical fashion, that is, a mediated schema can be defined on another mediated schema or on an exported source schema. The query language used by the Nimble system is XML-QL. When a query is posed to the integration system, it is decomposed into multiple source queries based on the data sources. The compiler translates each such

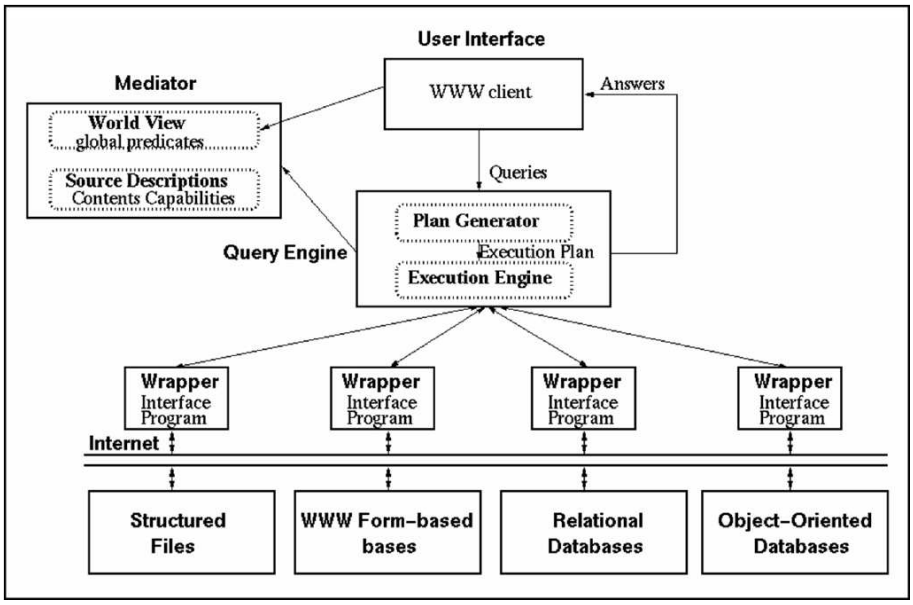


Figure 3.9: Information Manifold Architecture

query into the appropriate query language for the destination source.

3.2.6.2 LAV Systems

Information Manifold [KLSS95] [LRO96] [LSK95] handles the problem of data integration by providing a mechanism to describe declaratively the contents and the query capabilities of the information sources. The contents of the sources are expressed with the help of source descriptions, which are used efficiently by the system to prune the information sources that do not provide any answer to a user query. In Information Manifold system the global schema is relational. A source description is a conjunctive query over the global schema relations, which will be referred to as view. User queries, posed in Information Manifold, are conjunctive queries like source descriptions. They are expressed in terms of the global schema relations. Query rewriting is done by the Bucket algorithm, or an improved version of it, the MiniCon Algorithm, both discussed in Section 3.2.3. The architecture on Information Manifold is illustrated in Figure 3.9.

Infomaster [DG97] is an information system which provides integrated and uniform access to multiple distributed, heterogeneous, structured sources. It uses a three level

abstraction hierarchy for modeling the global schema and the sources. Data available in a source is also seen as a set of relations, called site relations. Between site relations and interface relations, a set of base relations is defined. Site relations are described as views over the base relations (following the LAV approach), while interface relations are defined as views on the base relations (following the GAV approach). User queries are expressed in terms of the interface relations and are firstly transformed in terms of the base relations (query unfolding following GAV approach). Then these queries are rewritten in terms of site relations using the Inverse Rules Algorithm (see Section 3.2.3.2).

Agora [MFK01] system supports querying and integration of heterogeneous relational and XML information sources. The global schema is an XML DTD and a virtual relational schema is used as an interface between the sources and this schema. Relational and XML resources are modeled as SQL queries over the relational global schema. Users formulate XQuery queries in terms of this global DTD. These queries are normalized and translated into an SQL query over the generic relational schema. Query rewriting is based on existing methods and on algorithms for rewriting queries using SQL views.

3.2.6.3 GLAV Systems

There are two basic representative systems for the GLAV approach: MARS and Dis@Dis.

MARS [DT03a] is a system for publishing as XML data from mixed (relational and XML) proprietary storage, while supporting redundancy in storage for tuning purposes. In this system XML and relational integrity constraints are taken into consideration. It accepts client XQueries formulated against the public schema and with the help of rewriting-with-views, composition-with-views and query minimization under integrity constraints it achieves optimal reformulation against the proprietary schema. To be able to combine GAV and LAV, MARS is switched to a different, "direction-neutral", representation of the views, compiling them into constraints. We further discuss MARS in Chapter 6 since MARS is a basic component

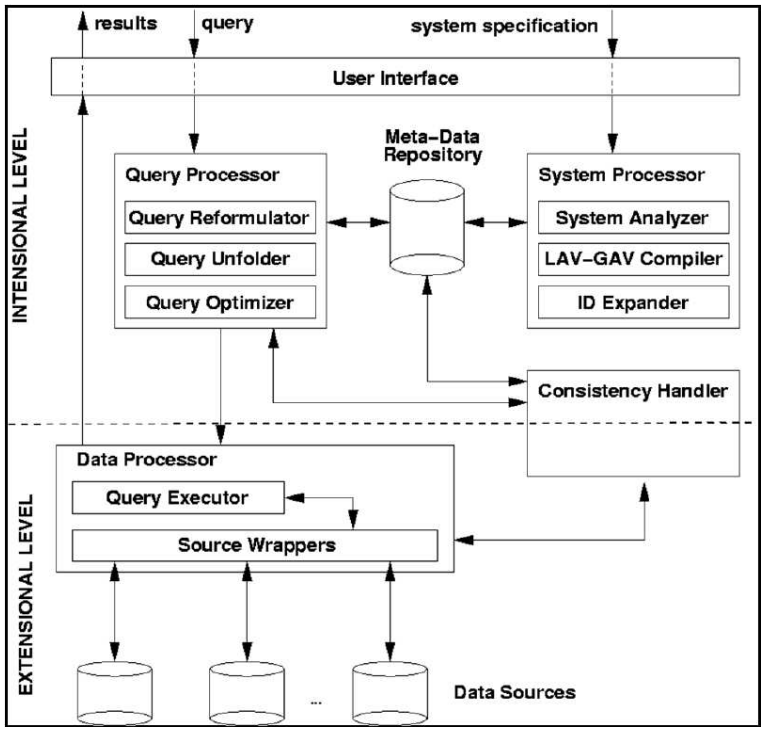


Figure 3.10: Dis@Dis Architecture

in the implementation of our integration system (SWIM).

Dis@Dis [CLR03] deals with the problem of query answering when data does not satisfy integrity constraints. This happens, for instance, in data integration, where integrity constraints that enrich the semantics of the global view are violated by data in the sources. Dis@Dis provides a thorough analysis of data integration (both in the GAV and in the LAV approach) in the presence of both key dependencies (KDs) and inclusion dependencies (IDs). More specifically, system results advance the state-of-the art in data integration in terms of: (i) decidability results for general or key-consistent database instances; (ii) new algorithm computing query answering in the presence of both KDs and cyclic IDs; (iii) a modular treatment of GAV and LAV systems in the presence of integrity constraints, through the compilation of LAV into GAV, and of KDs and IDs, through the separability property of non-key-conflicting (NKC) dependencies; (iv) effective techniques for data integration, which exploit both off-line and intensional processing of queries (this processing is independent of the data in the sources) (Figure 3.10). Experiments have shown that the intensional

processing of queries is actually performed very fast, despite the complexity of the tasks that are computed (which in the worst case require time exponential in the size of the query). Future Dis@Dis extensions include more complex forms of dependencies, e.g., functional and exclusion dependencies.

Chapter 4

The SWIM Logic Framework

Choosing an expressive logic framework for our data integration middleware, is crucial for the manipulation of the integrated data. Republishing XML (virtual or native) sources as RDF and querying them are situations that need to be dealt with, through a powerful framework. In SWIM, we have adopted a *first-order logic* representation of both XML and RDF. The advantages of such a choice emanate basically from the fact that background theory on relational query optimization can be exploited. Query containment and minimization, query composition and query rewriting using views are solvable for a fairly large class of queries in the presence of certain classes of constraints. Linear Datalog (i.e., non-recursive Datalog without negation) has been chosen as a robust formalism to rely on.

The choice of first-order logic leads us to find an appropriate representation for XML and RDF based on a set of relations. This representation is presented in the sequel.

4.1 Preliminary Definitions

Definition 4.1 *A dependency is a first-order logic assertion of the general form*

$$\forall x_1,\ \ldots,\ \forall x_n\ [\ \varphi(x_1,\ \ldots,\ x_n)\ \rightarrow\ \exists z_1,\ \ldots,\ \exists z_k \varphi'(y_1,\ \ldots,\ y_m)\ ]$$

where $\{z_1, \ldots, z_k\} = \{y_1, \ldots, y_m\} - \{x_1, \ldots, x_n\}$, φ *is a (possible empty) conjunction of atoms and* φ' *a nonempty conjunction. In both* φ *and* φ'*, one finds relation atoms of the form* $R(w_1, w_2, \ldots, w_l)$ *and equality atoms of the form* $w = w'$*, where each of the* $w_1, w_2, \ldots, w_l, w, w'$ *is a variable.*□

Definition 4.2 *An embedded dependency is a dependency (Definition 4.1) where:*

- φ *is a conjunction of relation atoms using all of the variables* $x_1, \ldots, x_n$
- φ' *is a conjunction of atoms using all of the variables* $z_1, \ldots, z_k$
- *there are no equality atoms in* φ' *involving existentially quantified variables.*□

Generalizing from embedded dependencies, we consider the class of disjunctive embedded dependencies [AHV95]:

Definition 4.3 *A disjunctive embedded dependency (DEDs) is an assertion of the form*

$$\forall x_1, \ldots, \forall x_n \, [\, \varphi(x_1, \ldots, x_n) \rightarrow \bigvee_i \exists z_1, \ldots, \exists z_k \varphi'_i(y_1, \ldots, y_m) \,] \; \square$$

4.2 The First-Order Logic Representation of XML

The first-order logic representation of the XML data model consists of the predicates capturing the XML constructs along with a set of constraints (DEDs) preserving XML semantics.

4.2.1 XML Basic Relations

XML documents and structures can be represented by a set of basic relations: XMLroot, T, A, Txt, C, D, E. For the representation of the XML document it is assumed that each node in the document is uniquely identified by a number, which is kept internally in the node. This set of relations is called **GReX** (**G**eneric **R**elational **e**ncoding of **X**ML).

- Relation XMLroot (for the root of the document):

name: **XMLroot**
e: int

This relation represents the root element node of an XML tree. `e` is the integer that identifies this root element.

- Relation T (for the element nodes):

name: **T**	
e: int	l: string

This relation represents the element nodes of an XML tree. `e` is the integer that identifies the node, which has a label `l`.

- Relation Txt (for the content of the element nodes):

name: **Txt**	
e: int	str: string

This relation represents the content of the element nodes of an XML tree. `e` is the integer that identifies a node, which has as content the string `str`.

- Relation A (for the attribute nodes):

name: **A**		
e: int	n: string	v: string

This relation represents the attribute nodes of an XML tree. `e` is the integer that identifies the node, which has a label `n` and its content (value) is `v`.

- Relation C (for the child relationship):

name: **C**	
s: int	t: int

This relation represents the parent - child relation between two nodes in the XML tree. `s` is the integer that identifies a node, which has a child with node identifier `t`.

- Relation D (for the descendant relationship):

name: **D**	
s: int	t: int

This relation represents the ancestor - descendant relation between two nodes in the XML tree. `s` is the integer that identifies a node, which has a descendant with node identifier `t`.

- Relation E (for the element nodes that are children of another node):

name: **E**		
s: int	t: int	l: string

This relation represents the child relation between two nodes in the XML tree and additionally identifies the child node. `s` is the integer that identifies a node, which has a child with node identifier `t` labeled `l`.

Example 4.1 *In order to make the above relations clearer we present how an XML tree is represented through these relations. Assume the XML tree shown in Figure 4.1.*

In this figure the XML nodes are represented by ovals, while there is a number (the identification number of the node) next to each oval. Each line (edge) represents a parent-child relationship, while the double line represents an ancestor - descendant

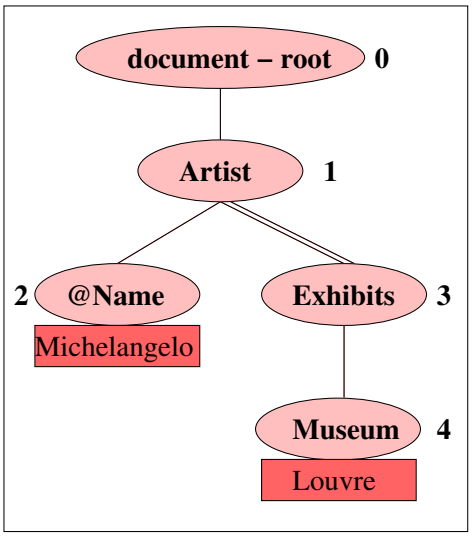


Figure 4.1: An example of an XML tree

relationship. Attribute nodes are distinguished by the prefix @ in their label. Additionally the box below a node represents the content of the node. The instances of the relations presented before for this XML tree are:

XMLroot
e
0

T	
e	***l***
3	*"Exhibits"*

Txt	
e	***str***
4	*"Louvre"*

E		
s	***t***	***l***
0	*1*	*"Artist"*
3	*4*	*"Museum"*

D	
s	***t***
1	*3*

A		
e	***n***	***v***
2	*"Name"*	*"Michelangelo"*

C	
s	***t***
1	*2*

□

4.2.2 XML Basic Constraints

There are some general constraints, which preserve the XML semantics. These constraints are disjunctive embedded dependencies defined in Definition 4.3.

In some constraints there are more than one different formulations. This is due to the fact that the relationship parent - child can be either through a C relation or an E relation. The identification of a node, also, can be either through a T or an E relation.

1. Every y that is a child of x implies that y is a descendant of x (*base*)

$$\forall x, y[C(x,\ y) \rightarrow D(x,\ y)]$$

or

$$\forall x, y, l[E(x,\ y,\ l) \rightarrow D(x,\ y)]$$

2. Every y that is a descendant of x and every z that is a descendant of y implies that z is a descendant of x (*transitivity*)

$$\forall x, y, z[D(x,\ y) \wedge D(y,\ z) \rightarrow D(x,\ z)]$$

3. Every node is a descendant of itself (*reflexivity*)

$$\forall x, l[T(x,\ l) \rightarrow D(x,\ x)]$$

or

$$\forall x, y, l[E(x,\ y,\ l) \rightarrow D(y,\ y)]$$

or

$$\forall x, n, v[A(x,\ n,\ v) \rightarrow D(x,\ x)]$$

4. For every two nodes x, y such that x is a descendant of y and y is a descendant of x, x and y are the same node (*anti-symmetry*)

$$\forall x, y[D(x,\ y) \wedge D(y,\ x) \rightarrow x = y]$$

5. For every three nodes x, y, z such that z is child of x and z is child of y, x and y are the same node (*one parent*)

$$\forall x, y, z[C(x,\ z) \wedge C(y,\ z) \rightarrow x = y]$$

or

$$\forall x, y, z, l[E(x,\ z,\ l) \wedge E(y,\ z,\ l) \rightarrow x = y]$$

or

$$\forall x, y, z, l[C(x,\ z) \wedge E(y,\ z,\ l) \rightarrow x = y]$$

or

$$\forall x, y, z, l[E(x,\ z,\ l) \wedge C(y,\ z) \rightarrow x = y]$$

6. Every y that is a child of x implies that both x and y are nodes (*nodes*)[1]

$$\begin{aligned}\forall x, y[C(x, y) \rightarrow \quad & \exists z, k, l_1, l_2, v\ [[T(x, l_1) \wedge T(y, l_2)] \\ & \vee [T(x, l_1) \wedge E(k, y, l_2)] \\ & \vee [E(z, x, l_1) \wedge T(y, l_2)] \\ & \vee [E(z, x, l_1) \wedge E(k, y, l_2)] \\ & \vee [E(z, x, l_1) \wedge A(y, l_2, v)] \\ & \vee [T(x, l_1) \wedge A(y, l_2, v)]]]\end{aligned}$$

7. Every y that is descendant of x implies that both x and y are nodes (*nodes*)

$$\begin{aligned}\forall x, y[D(x, y) \rightarrow \quad & \exists z, k, l_1, l_2, v\ [[T(x, l_1) \wedge T(y, l_2)] \\ & \vee [T(x, l_1) \wedge E(k, y, l_2)] \\ & \vee [E(z, x, l_1) \wedge T(y, l_2)] \\ & \vee [E(z, x, l_1) \wedge E(k, y, l_2)] \\ & \vee [E(z, x, l_1) \wedge A(y, l_2, v)] \\ & \vee [T(x, l_1) \wedge A(y, l_2, v)]]]\end{aligned}$$

8. Every node has one tag at most (*one tag*)

$$\forall x, l_1, l_2[T(x,\ l_1) \wedge T(x,\ l_2) \rightarrow l_1 = l_2]$$

or

$$\forall x, y, l_1, l_2[E(y,\ x,\ l_1) \wedge E(y,\ x,\ l_2) \rightarrow l_1 = l_2]$$

or

$$\forall x, y, l_1, l_2[E(y,\ x,\ l_1) \wedge T(x,\ l_2) \rightarrow l_1 = l_2]$$

[1]Node x cannot be an attribute node.

9. Every attribute has one value at most (*one attribute*)

$$\forall x, n, v_1, v_2[A(x,\ n,\ v_1) \wedge A(x,\ n,\ v_2) \rightarrow v_1 = v_2]$$

10. There is only one root node (*one root*)

$$\forall x, y[XMLroot(x) \wedge XMLroot(y) \rightarrow x = y]$$

11. Every y that is root and is descendant of x implies that x is root too (*top root*)

$$\forall x, y[D(x,\ y) \wedge XMLroot(y) \rightarrow XMLroot(x)]$$

12. Every z that is descendant of both x and y means that x and y have a descendant relationship or are the same node (*in line*)

$$\forall x, y, z[D(x,\ z) \wedge D(y,\ z) \rightarrow x = y \vee D(x,\ y) \vee D(y,\ x)]$$

13. Every y that is a child of x and there is a node z between them, means that z is the same node with either x or y (*choice*)

$$\forall x, y, z[C(x,\ y) \wedge D(x,\ z) \wedge D(z,\ y) \rightarrow x = z \vee y = z]$$

or

$$\forall x, y, z, l[E(x,\ y,\ l) \wedge D(x,\ z) \wedge D(z,\ y) \rightarrow x = z \vee y = z]$$

This set of constraints is inherent in the XML data model and is called **TIX** [Deu02]. TIX is an acronym for **T**rue **I**n **X**ML.

4.3 The First-Order Logic Representation of RDF

In a similar way the first-order logic representation of RDF consists of a set of relations and a set of constraints preserving the RDF semantics.

4.3.1 RDF/S Basic Relations

RDF/S schema classes and properties can be represented by the basic relations (CLASS, C_SUB, P_SUB, PROP, P_EXT, C_EXT):

- Relation CLASS:

name: **CLASS**
name: Class

This relation represents the RDF/S classes. `name` is the name of class and it is of type Class[2].

- Relation C_SUB:

name: **C_SUB**	
subC: Class	clas: Class

This relation represents the subclass hierarchy between two classes. Class `subC` is a subclass of the class `clas`.

- Relation P_SUB:

name: **P_SUB**	
subP: Property	prop: Property

This relation represents the subproperty hierarchy between two properties. Property `subP` is a subproperty of the property `prop`.

- Relation PROP:

[2]Class type, Property type and Resource type are in fact strings. However, we write them as Class, Property and Resource in order to help the reader to understand the relational schema.

name: **PROP**		
source: Class	name: Property	dest: Class

This relation represents the properties. Property with predicate `name` has as domain (subject) the class `source` and as range (object) the class `dest`.

- Relation C_EXT:

name: **C_EXT**	
clas: Class	inst: Resource

This relation represents the extension of each class. Resource `inst` belongs to the extension of the class `clas`.

- Relation P_EXT:

name: **P_EXT**		
s_inst: Resource	name: Property	d_inst: Resource

This relation represents the properties extension. There is an instance of property called `name` that has the resource `s_inst` as domain and the resource `d_inst` as range.

Example 4.2 *This example illustrates the use of the above relations in the representation of the RDF/S that is shown in Figure 4.2.*

CLASS
name
Artist
Artifact
Painter
Painting

PROP		
source	***name***	***dest***
Artist	*creates*	*Artifact*
Painter	*paints*	*Painting*
Artist	*name*	*string*
Artifact	*title*	*string*

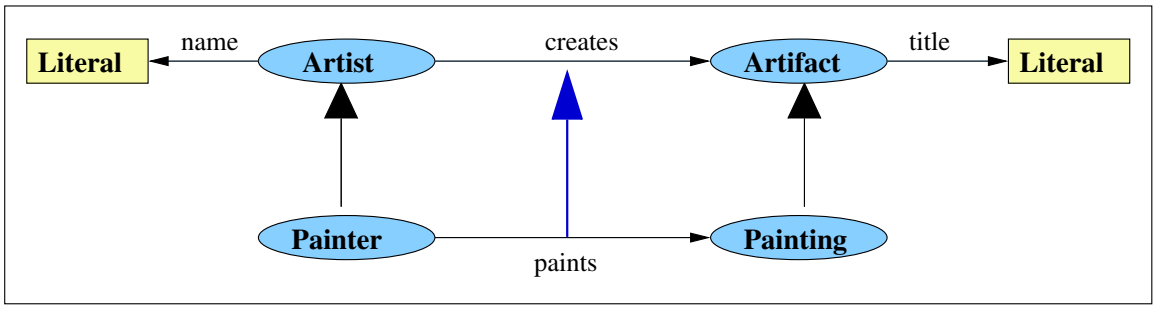


Figure 4.2: An example of an RDF schema

C_SUB	
subC	***class***
Painter	*Artist*
Painting	*Artifact*

P_SUB	
subP	***prop***
paints	*creates*

In this example, we do not present the relations capturing RDF instances (P_EXT, C_EXT), for the sake of simplicity. These instances are not presented in Figure 4.2 either.□

4.3.2 RDF/S Basic Constraints

The basic constraints used for preserving the semantics of RDF/S are:

1. Every resource in the extent of a class implies the existence of the corresponding class

$$\forall c, x[C_EXT(c,\ x) \rightarrow CLASS(c)]$$

2. Every statement in the extent of a property implies the existence of the corresponding property

$$\forall x, p, y[P_EXT(x,\ p,\ y) \rightarrow \exists c, dPROP(c,\ p,\ d)]$$

3. Every subclass of a class is also a class

$$\forall c_1, c_2[C_SUB(c_1,\ c_2) \rightarrow CLASS(c_1) \wedge CLASS(c_2)]$$

4. Every subproperty of a property is also a property

$$\forall p, q[P_SUB(p, q) \rightarrow \exists a, b, c, d[PROP(a,\ p,\ b) \wedge PROP(c,\ q,\ d)]]$$

5. The domain and range of every property is a class

$$\forall c_1, c_2, p[PROP(c_1,\ p,\ c_2) \rightarrow CLASS(c_1) \wedge CLASS(c_2)$$

6. The domain and range of every property is unique

$$\forall p, a_1, a_2, b_1, b_2[[PROP(a_1,\ p,\ a_2) \wedge PROP(b_1,\ p,\ b_2)] \rightarrow a_1 = a_2 \wedge b_1 = b_2]$$

7. Every class is a subclass of itself

$$\forall c[CLASS(c) \rightarrow C_SUB(c,\ c)]$$

8. The subclass relationship is transitive

$$\forall c_1, c_2, c_3[[C_SUB(c_1,\ c_2) \wedge C_SUB(c_2,\ c_3)] \rightarrow C_SUB(c_1,\ c_3)]$$

9. Every property is a subproperty of itself

$$\forall a, p, b[PROP(a,\ p,\ b) \rightarrow P_SUB(p,\ p)]$$

10. The subproperty relationship is transitive

$$\forall p_1, p_2, p_3[[P_SUB(p_1,\ p_2) \wedge P_SUB(p_2,\ p_3)] \rightarrow P_SUB(p_1,\ p_3)]$$

11. In a valid RDF description schema the domain (range) of a subproperty is subsumed by the domain (range) of a super-property

$$\forall a, p, b, c, q, d[[P_SUB(q,\ p) \wedge PROP(a,\ p,\ b) \wedge PROP(c,\ q,\ d)] \rightarrow [C_SUB(c,\ a) \wedge C_SUB(d,\ b)]]$$

12. In a valid RDF description base the subject/object resources in every statement are instances of the property's domain/range classes (either direct or subclass instances)

$$\forall a, p, b, x, y[[PROP(a,\ p,\ b) \wedge P_EXT(x,\ p,\ y)] \rightarrow$$
$$\exists c, d[C_SUB(c,\ a) \wedge C_SUB(d,\ b) \wedge C_EXT(c,\ x) \wedge C_EXT(d,\ y)]]$$

The constraints above can be classified into the following general categories: Constraints 1 to 6 are *basic* RDF/S constraints. Constraints 7-10 are the *SUB* constraints. This means that they focus on the properties of the subclass - subproperty relationship. The last two constraints (11, 12) are the *domain - range* constraints which focus on some special properties of the domain (range) of the RDF/S properties.

Let Δ_{RDF} be the set of the dependencies used to capture the internal RDF/S data model. It has been proven and thoroughly discussed in [Ser05] that:

Theorem 4.1 *It is decidable whether* $\Delta_{RDF} \models d$ *and whether* $\Delta_{RDF} \models Q_1 \sqsubseteq Q_2$, *where d is an embedded implicational dependency,* $Q_1,\ Q_2$ *are conjunctive queries and* $\sqsubseteq$ *is query containment.* ■

4.4 Differences with the RDF/S Model Theory

There are some differences between the SWIM logic framework (SWLF) semantics and the original RDF/S model theory [rdfb] [rdfa]. These differences stem from the assumption made by the SWLF (and additionally the RQL) which declares that there are three levels of abstraction (data, schema and metaschema). This is not stated declaratively in the RDF/S model theory. This assumption imposes the following constraints: A class can be an instance of a metaclass and of no other schema-level class. That is, between two classes at schema-level only a subsumption relationship can be defined. Additionally, a class can not subsume a metaclass. Subsumption is defined only in the schema and metaschema levels of abstraction and not between them. Finally, a metaclass cannot be an instance of something else, since there are only three levels of abstraction. For simplicity, SWIM does not model metaschema information.

Another difference between the SWLF semantics and the RDF/S model theory is the fact that the former imposes the constraint of a *defined* and *unique* domain and

range for every property. This is not an hypothesis made by the RDF/S semantics, since they assume that a property may (not mandatorily) have one or more domains (ranges). In case of an undefined property's domain (range) this can be instantiated by both resources and literals. This can lead to semantic inconsistencies.

Furthermore, SWLF imposes one more restriction in the subproperty hierarchy. The fact that a property subsumes another, supposes that the domain (range) class of the first subsumes the domain (range) class of the second. Finally, the cyclic subsumption between classes and properties is not supported. This, in accordance to the use of multiple namespaces could affect the semantics of already defined RDF/S schemas.

Chapter 5

Query Reformulation in SWIM

SWIM is the basic component of a data integration system for publishing XML and relational sources [Ser05] on the Semantic Web. User queries are expressed in terms of a virtual RDF/S schema and are in turn reformulated in terms of the target XML sources in an XML query language (XPath, XQuery). As discussed in Chapter 3, this query reformulation raises many questions. In this chapter we address (i) the SWIM mapping language employed to reconciliate RDF/S schemas and XML documents and (ii) the SWIM query reformulation algorithm which is based on these mappings and achieves the translation of an RQL query into an XPath and/or XQuery query.

5.1 From XML Data to RDF/S Schema

RDF/S and XML rely on two different data models. RDF/S, as presented in Section 2.2, is represented by a *labeled graph* that is a graph with labeled nodes and labeled edges (arcs). Moreover, its edges are directed. On the contrary the XML data model is a *labeled tree*, with labeled nodes (only) and undirected edges. Each label on the nodes of the XML tree represents the tags of an XML document and the parent-child relationship reenacts the nesting structure of the document. While in RDF/S there is no ordering, in XML the children of a node are ordered. The question that, naturally, arises from these differences is how we can use our mapping rules in

order to reconciliate these data models.

At this point, we should declare the differences between *correspondences* and *mappings* [Myl04]. Formally, correspondences constitute binary relations between symbols while, mappings define translations from expressions defined with respect to one model to expressions defined with respect to another model.

In SWIM we foresee the following correspondences of the XML to the RDF/S data model:

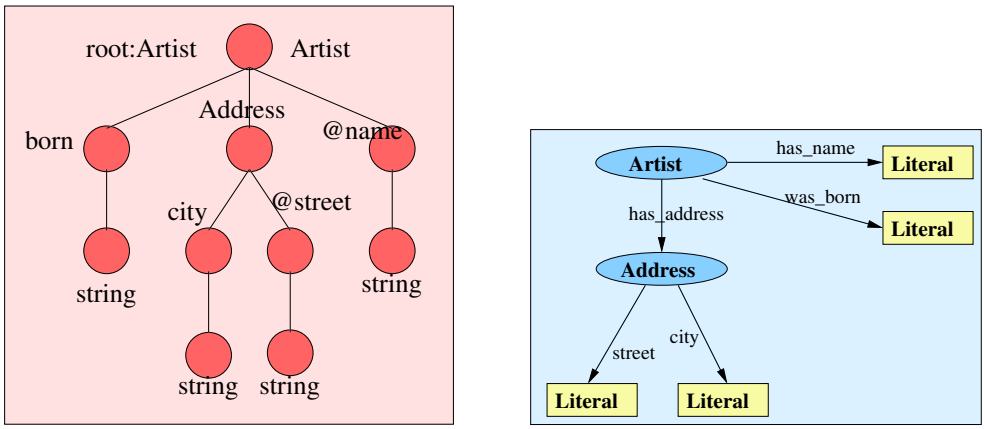
- Each XML tag corresponds to either an RDF/S class (which means that there is a unique URI for the resources of this class) or to a literal node.
- XML attribute nodes or XML textual elements correspond to RDF/S properties with range node of type literal. All the other element nodes correspond to RDF/S classes.
- XML complex elements (elements with children) are *additionally* mapped to RDF/S properties with range of type RDF/S class.

The mappings above will be declared through an example:

Example 5.1 *Let's consider the following DTD that constitutes the grammar of an XML document.*

```
<!DOCTYPE Artist [
<!ELEMENT Artist (born, Address?)>
<!ATTLIST Artist name CDATA #REQUIRED>
<!ELEMENT born (#PCDATA)>
<!ELEMENT Address (city)>
<!ATTLIST Address street CDATA #REQUIRED>
<!ELEMENT city (#PCDATA)>
]>
```

This DTD can be represented by an XML tree as shown in Figure 5.1(a). Figure 5.1(b) represents an RDF/S schema. The rules presented above, will lead to the following correspondences:



(a) The XML tree

(b) The RDF graph

Figure 5.1: Defining correspondences between XML and RDF schema

- *XML tags* **`Artist`** *and* **`Address`** *correspond to the RDF/S classes* **`Artist`** *and* **`Address`** *respectively.*

- *XML attribute nodes* **`name`** *and* **`street`** *correspond to the properties* **`has_name`** *and* **`street`** *that have the classes* **`Artist`** *and* **`Address`** *respectively as domain, and their range node is of type literal.*

- *XML textual element nodes like* **`born`** *and* **`city`** *correspond to the properties* **`was_born`** *and* **`city`** *that have the classes* **`Artist`** *and* **`Address`** *respectively as domain, and their range node is of type literal.*

- *XML complex element* **`Address`** *which is a child of* **`Artist`** *corresponds to the property* **`has_address`** *with domain the RDF/S class* **`Artist`** *and range the RDF/S class* **`Address`**.

□

5.2 SWIM Mapping Rules

The correspondences defined previously must be founded formally. As explained before (Chapter 4), Linear Datalog is a well-known framework and will be quite useful for defining mapping rules.

Definition 5.1 *The general form of a mapping rule is:*

$$\phi_{RVLclause}(\overline{\chi}) \ :- \ \phi_{XBind}(\overline{\chi}')$$

where $\phi_{RVLclause}(\overline{\chi})$ is a conjunction of view clauses employed by the RDF/S view definition language RVL, $\phi_{XBind}(\overline{\chi}')$ is a conjunction of path expressions (called XBind queries), and $\overline{\chi} \subseteq \overline{\chi}'$.□

The conjunction of the view clauses appearing at the head of the Datalog rules consists of clauses in the form $A(\chi)$ or $A'(\chi, \psi)$, where $A(\chi)$ defines the direct extent of class A in the virtual RDF layer and $A'(\chi, \psi)$ defines the extent of property A'. These RVL atoms can be translated to the RDF SWLF predicates (C_EXT, P_EXT) according to the following rules:

1. each clause in the form $A(\chi)$ can be translated into the predicate C_EXT(A, χ), while
2. each clause in the form $A'(\chi,\ \psi)$ can be translated into P_EXT($\chi,\ A',\ \psi$).

The body of the rule consists of XPath expressions, instead of the XML SWLF predicates, in order to present the XML in a shorter form. The translation of the XPath expressions into the XML SWLF predicates (XMLroot, A, T, E, C, D, Txt), is harder than the one of RVL clauses into RDF SWLF predicates. Each path should be translated into a set of appropriate predicates (see Example 5.2). Furthermore, there are equations or built-in predicates in the body.

XPath atoms are used for variable binding and can be unary or binary [Deu02]. In other words, they can be followed by one or two variables. An absolute path leads to a node whose content is bound to the single variable that follows such a path. Each relative path is followed by two variables: the first represents the start node of the path and the second the content of the final node of the path.

Example 5.2 *The predicate {/A/B[D=3]/C}(x) binds variable x to the content of node C, while {.//E/F}(x,y) binds the variable y to the content of the child (with tag name F) of a node with tag name E that is a descendant of a node that has been*

bound to variable x. The first path should be translated into:
r in XMLroot, e1 in E, e2 in E, e3 in E, e4 in E, txt1 in Txt , txt2 in Txt
*where (e1.s=r) and (e1.l="A") and (e2.s=e1.t) and (e2.l="B") and (e3.s=e2.t) and (e3.l="D") and (e4.s=e2.t) and (e4.l="C") and (txt1.e=e3.t) and (txt1.str=3) and (txt2.e=e4.t) and (txt4.str=**x**).*
The second one should be translated into[1]*:*
d1 in D, e1 in E e2 in E, txt1 in Txt
*where (d1.s=**x**) and (e1.s=d1.t) and (e1.l="E") and (e2.s=e1.t) and (e2.l="F") and e2.t=txt1.e) and (txt1.str=**y**).*□

Additionally, the framework for the definition of the mapping rules is flexible enough to include some relational atoms that serve in handling more intricate situations. For example, we employ non interpreted functions (e.g., for handling strings) in order to establish more expressive mappings (e.g., involving complex keys). These functions are not interpreted as Datalog functions. They are first-order logic predicates defined in the SWIM logical framework and their use resembles to the use of the other predicates. They are handled during the query translation phase (discussed in Section 5.5) and not during the reformulation phase (discussed in Section 5.3). For example, the `Concat(x, y, z)` atom represents the construction of a string `x` by concatenating the strings `y` and `z`, while the `Split(a, b, x, y)` atom represents the splitting of the string `x` into two strings `a` and `b` at the first appearance of the delimiter `y`.

The BNF grammar that can represent the general syntax of the mapping rules is depicted in Table 5.1.

Example 5.3 *Given the Example 5.1 we can define the following mappings (they are not the only ones):*

[1]The interpretation of a//b (according to the semantics of XPath2.0) is: a/descendant-or-self::node/child::b.

[1]	rule	::=	head *":-"* body.
[2]	head	::=	view_clause+
[3]	view_clause	::=	class_extension | property_extension
[4]	body	::=	xpath_statement+ | equality* (*"|"* function+)?
[5]	xpath_statement	::=	*"{"* xpath *"}"* *"("* variables *")"*
[6]	equality	::=	variable *"="* variable | variable *"="* value
[7]	xpath	::=	BNF Table 2.2
[8]	function	::=	concat_function | split_function | ...
[9]	variables	::=	variable | variable *","* variable

Table 5.1: Simplified BNF Grammar for Mapping Rules

Artist(x) :- *{//Artist}(k), {./@name}(k, x).*

Address(x) :- *{//Address}(k), {./@street}(k, x).*

has_address(x, y) :- *{//Artist}(k), {./@name}(k, x),*
{./Address}(k, a), {./@street}(a, s),
{./city}(a, t) | concat(y, s, t).

□

5.2.1 Capturing Mappings through Constraints

As has been mentioned, SWIM middleware doesn't adopt one of the main approaches used for data integration (LAV and GAV) but considers a hybrid one: GLAV approach. This is accomplished by representing the mapping rules as constraints.

Definition 5.2 *Given the formal Definition 3.11, the mapping between the global schema and the sources, constitutes of the following assertions:*

$G \supseteq S$ *(sound source)*

or

$G \subseteq S$ *(complete source)*

or

$G \equiv S$ *(exact source)*

where G is the global schema, and S is the local one.

The soundness of the mappings is ensured by the interpretation of each mapping as:

$$\forall \overline{\chi},\ \overline{\psi}\ [\phi_{XBind}(\overline{\chi},\ \overline{\psi})\ \rightarrow\ \phi_{RVLclause}(\overline{\chi})] \qquad (1)$$

while the completeness is obtained by the (additional) interpretation of each mapping as:

$$\forall \overline{\chi}\ [\phi_{RVLclause}(\overline{\chi})\ \rightarrow\ \exists\ \overline{\psi}\ \phi_{XBind}(\overline{\chi},\ \overline{\psi})] \qquad (2)$$

□

Given the Definition 5.2 it is stated that the mappings between the global *schema* and the local ones can be either sound, complete or exact. In the general case, the GLAV approach models a situation where the sources are sound [Len02]. However, SWIM's approach assumes *only* exact mappings. This means that a query addressed to the global schema will return *all* and the *correct* results.

The assumption of only exact mappings is realized from the fact that in our mapping rules (Definition 5.1) we require that $\overline{\chi}\ \subseteq\ \overline{\chi}'$. If this requirement is not satisfied exactness **is not** preserved. This is due to the fact that in this case, the second (2) constraint will be:

$$\forall \overline{\chi}\ [\phi_{RVLclause}(\overline{\chi})\ \rightarrow\ \exists\ \overline{\psi}\ \phi_{XBind}(\overline{\chi}',\ \overline{\psi})] \qquad (2')$$

and $\overline{\chi} \nsubseteq \overline{\chi}' \cup \overline{\psi}$.

Figure 5.2 illustrates the interpretation of the mapping by two constraints.

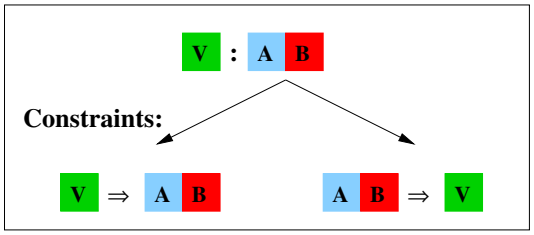


Figure 5.2: Mapping interpretation to constraints

The constraint on the left part of the Figure 5.2 means that all data appearing in "V" satisfy "A" and "B", while the constraint on the right part means that all data satisfying "A" and "B" appear in the result "V".

The above interpretation of the mappings follows the GLAV approach of SWIM, since in Definition 5.2 the first constraint (1) gives the GAV approach of the mappings, while the second (2) gives the LAV one.

Example 5.4 *The mapping defined for Artist(x) in Example 5.3 is interpreted by the following two constraints.*

$$\forall\ x\ [Artist(x)\ \rightarrow$$
$$\exists\ a, m, s, k, v\ XMLroot(a),\ D(a, m),\ E(m, s, \text{“}Artist''),\ C(s, k),\ A(k, \text{“}name'', v),$$
$$x = v]$$
$$and$$
$$\forall\ a, m, s, k, v\ [XMLroot(a),\ D(a, m),\ E(m, s, \text{“}Artist''),\ C(s, k),$$
$$A(k, \text{“}name'', v)\ \rightarrow\ \exists\ x\ Artist(x),\ x = v]$$

□

Another issue that should be discussed is the fact that the mapping rules where the head is a conjunction of more than one RVL clauses, can provide some information for each of the RVL clauses separately. This may prove very useful in cases where there do not exist distinct mappings for some of the RVL clauses.

Example 5.5 *Let's assume the following mapping:*

Sculptor(x),school(x,y) :- *{//Sculptor}(k), {./@name}(k, x),*
{./school}(k, y).

This mapping rule will be translated to the constraints:

$$\forall\ x, y\ [Sculptor(x), school(x, y)\ \rightarrow$$
$$\exists\ a, m, s, t, k, v, v1\ XMLroot(a),\ D(a, m),\ E(m, s, \text{“}Sculptor\text{”}),\ C(s, k),$$
$$A(k, \text{“}name\text{”}, v),\ E(s, t, \text{“}school\text{”}),\ Txt(t, v1),\ x = v,\ y = v1]$$
$$and\ (the\ inverse\ constraint)$$
$$\forall\ a, m, s, t, k, v, v1\ [XMLroot(a),\ D(a, m),\ E(m, s, \text{“}Sculptor\text{”}),\ C(s, k),$$
$$A(k, \text{“}name\text{”}, v),\ E(s, t, \text{“}school\text{”}),\ Txt(t, v1)\ \rightarrow$$
$$\exists\ x, y\ Sculptor(x),\ school(x, y),\ x = v,\ y = v1]$$

Additionally, we can infer the constraints:

$$\forall\ x\ [Sculptor(x)\ \rightarrow$$
$$\exists\ a, m, s, t, k, v, v1\ XMLroot(a),\ D(a, m),\ E(m, s, \text{“}Sculptor\text{”}),\ C(s, k),$$
$$A(k, \text{“}name\text{”}, v),\ E(s, t, \text{“}school\text{”}),\ Txt(t, v1),\ x = v]$$
$$and$$
$$\forall\ x, y\ [school(x, y)\ \rightarrow$$
$$\exists\ a, m, s, t, k, v, v1\ XMLroot(a),\ D(a, m),\ E(m, s, \text{“}Sculptor\text{”}),\ C(s, k),$$
$$A(k, \text{“}name\text{”}, v),\ E(s, t, \text{“}school\text{”}),\ Txt(t, v1),\ x = v,\ y = v1]$$

(The inverse constraints of the constraints presented above are not needed due to the existence of the previous inverse constraint.) □

5.2.2 The XPath Fragment used for Mappings

XPath 2.0 is used for the expression of SWIM mappings. As it has been declared every predicate in the body of a mapping can be either an equation, a built-in predicate or a path that binds some variables. Mapping rules use a small fragment of XPath 2.0.

The simple BNF defined for the mappings (Table 5.1) uses a small fragment of the XPath 2.0 BNF, presented in Section 2.1.2. This fragment is:

[1]	(simple xpath) p	::=	$p_1 \mid p_2 \mid /p \mid //p \mid p_1/p_2$ $\mid p_1//p_2 \mid p[q] \mid . \mid n \mid @n \mid @*$ $\mid \$u := @n \mid \texttt{text}() \mid \texttt{id}(p) \mid \texttt{id}(s)$
[2]	(qualifier) q	::=	q_1 `and` $q_2 \mid q_1$ `or` $q_2 \mid p \mid p = s \mid @n = \$u \mid \$u_1 = \$u_2$

Table 5.2: Simple XPath Grammar

where `n` is any tag or attribute name, `u` is any variable and `s` is any string constant. For this simple fragment of XPath it has been proven [DT01] that:

Theorem 5.1 *Containment of simple XPath expressions (no constraints) is* $\prod_2^p$ *- hard. Containment of disjunction-free simple XPath expressions (no constraints) is* NP *- hard.* ■

The used fragment of XPath includes all the navigational axes, except for those concerning sibling navigation (preceding, preceding - sibling, following, following -

sibling). These axes are not contained since in our first-order logic representation of XML documents, order is disregarded.

This fragment of XPath also includes the abbreviated form of navigation, instead of the full path. For example, we can write a/b//c instead of a/child::b/descendant-or-self::node()/child::c.

One more restriction on XPath 2.0 is that we do not use functions or expressions with for, let, if etc or equalities between paths. Additionally, we disallow navigation steps via the child axis from or to elements of unspecified tag. This can be done by using either the wildcard `*` for foregoing to a child of unspecified tag name, the `parent` axis to get to a parent of unspecified tag name or the `ancestor` axis without specifying the tag name. The only navigation step of unspecified tag that is permitted is the one of wildcard attribute tag (.../@*). In [DPT01] has been proven that:

Theorem 5.2 *Adding (i) path equality or (ii) navigation steps of unspecified tag to disjunction-free simple XPath expressions makes the containment problem (no constraints)* $\prod_2^p$ *-hard.* ■

5.2.3 Expressiveness of the SWIM Mappings

The mappings, presented above, are a powerful tool for expressing the correspondences between the XML documents and the RDF/S schema. The expressiveness of the mappings is based on the fact that we can establish correspondence between any part of the XML data model (part of a labeled tree) and any part of the RDF/S data model (part of a labeled graph). In a tree, which represents an XML document, there are *edges*, *paths*, or even *subtrees*. Respectively, in a graph, which represents an RDF/S schema, there are *edges*, *paths* and generally *subgraphs*. Our mapping rules are capable of matching these different "parts".

In order to declare the above assertions, let's consider the following example (Example 5.6), that captures some representative cases of mappings. For each case, we show the graphical presentation of the fragments (of both XML and RDF/S) that should be mapped to one another.

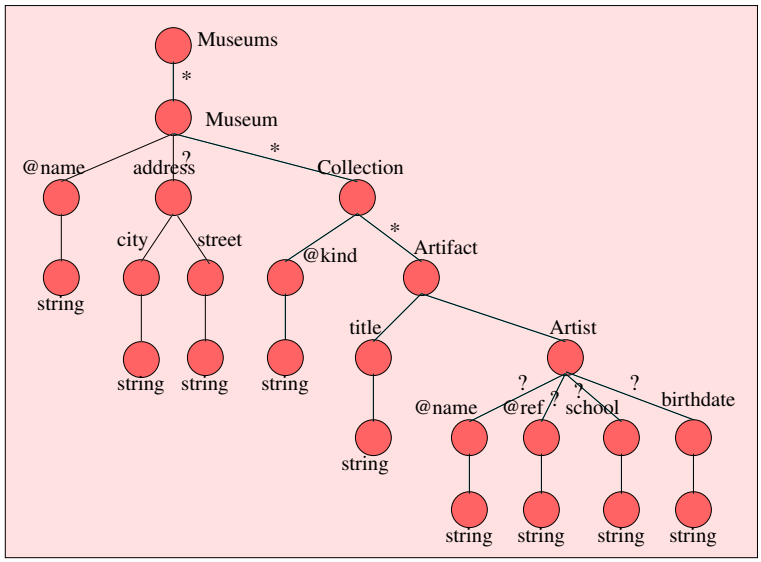


Figure 5.3: The tree representation of the XML structure

Example 5.6 *Let's assume that the XML source can be described by the following DTD:*

```
<!DOCTYPE Museums [
<!ELEMENT Museums (Museum*)>
<!ELEMENT Museum (address?, Collection*)>
<!ATTLIST Museum name ID #REQUIRED>
<!ELEMENT Collection (Artifact*)>
<!ATTLIST Collection kind (painting|sculpture) ‘‘painting’’>
<!ELEMENT address (city, street)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT street (#PCDATA)>
<!ELEMENT Artifact (title, Artist)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT Artist (school?, birthdate?)>
<!ATTLIST Artist name ID #IMPLIED>
<!ATTLIST Artist ref IDREF #IMPLIED>
<!ELEMENT school (#PCDATA)>
<!ELEMENT birthdate (#PCDATA)> ]>
```

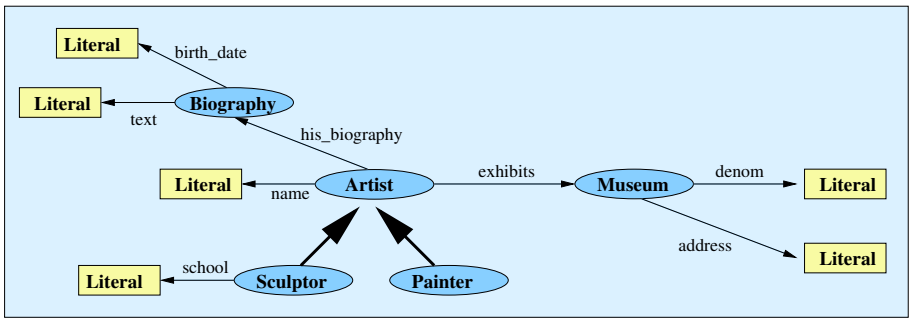
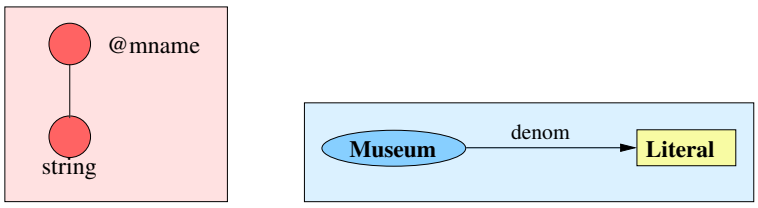


Figure 5.4: The RDF/S schema

Figure 5.3 depicts the graphical representation of this DTD. The RDF/S virtual schema used for the integration is shown in Figure 5.4. Some cases of mappings we consider are the following:

1. *The simplest mapping we can specify is between an XML edge and an RDF/S edge. Let's assume that the name of a* `Museum` *is called* `mname` *instead of the* `name` *that appears in our DTD. This attribute mname can be graphically represented as shown in Figure 5.5(a). The RDF/S correspondent of this relation is the graph shown in Figure 5.5(b) and the mapping is expressed by the rule:*

$$denom(x,\ y) \quad :- \quad \{//@mname\}(x),\ x{=}y.$$



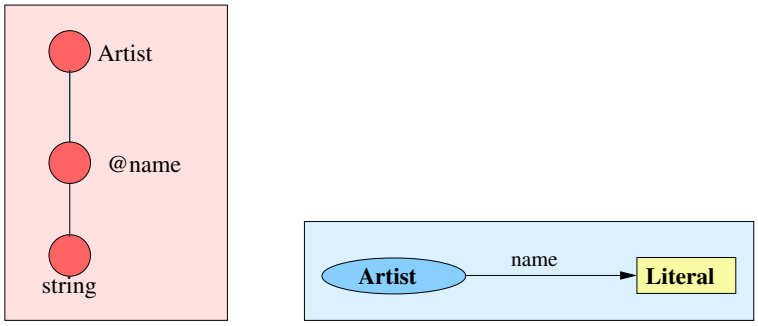
(a) An XML edge (b) An RDF/S edge

Figure 5.5: Mapping an XML edge to an RDF/S property

2. *More complex mappings arise when we need to establish a correspondence between an XML path (Figure 5.6(a)) and an RDF/S property.*

In this particular case let's assume the relationship between the `Artist` *and his* `name`*, which is shown in Figure 5.6(a). Figure 5.6(b) depicts the correspondent*

relationship of the RDF/S schema. Accordingly, the mapping between the XML path and the RDF/S edge, will be:

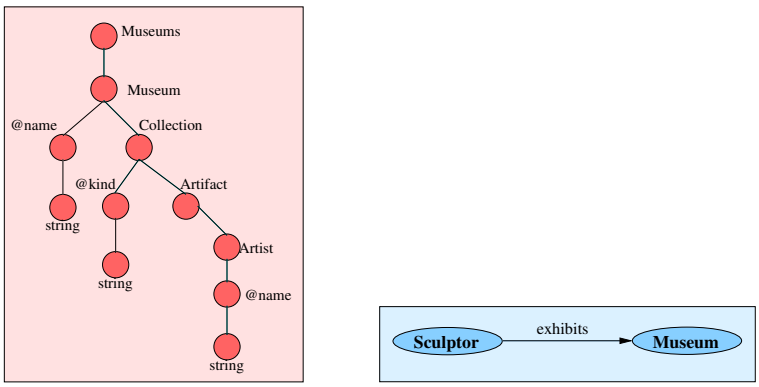


(a) An XML path (b) An RDF/S edge

Figure 5.6: Mapping an XML path to an RDF/S property

$$name(x,\ y) \quad :- \quad \{//Artist/@name\}(x),\ x{=}y.$$

3. *Mapping can be more sophisticated when we need to a establish correspondence between an entire XML subtree and an RDF/S edge. In this example we want to represent the fact that an* `Artist` *(and especially a* `Sculptor`*) exhibits in* `Museums` *(Figure 5.7(a)).*



(a) An XML subtree (b) An RDF/S edge

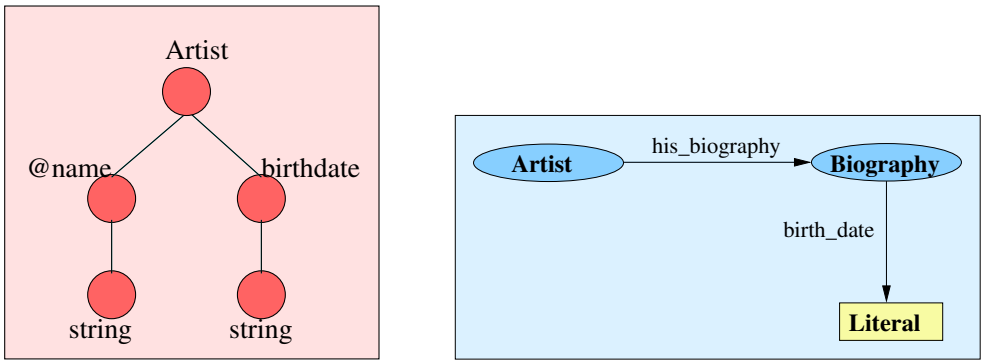
Figure 5.7: Mapping an XML tree to an RDF/S property

The RDF/S correspondent of the relationship between `Sculptor` *and* `Museum` *is the graph shown in Figure 5.7(b).*[2] *Therefore the mapping will be:*

[2]Since `Sculptor` is a subclass of `Artist` it inherits the `Artist's` property `exhibits`.

exhibits(x, y) :- {*//Museum*}(*a*), {*./Collection*}(*a,k*), {*./@kind*}(*k, s*),
{*.//Artist/@name*}(*k, x*),{*./@name*}(*a,y*),
s = "Sculpture".

This is the case when an XML tree is mapped to an RDF/S edge.



(a) An XML subtree (b) An RDF/S path

Figure 5.8: Mapping an XML tree to an RDF/S path

4. *The most expressive form of mappings implies a correspondence between an XML subtree and an RDF/S path or subgraph. In this last example we represent the relationship between an* `Artist` *and his* `birthdate` *(Figure 5.8(a)). The RDF/S correspondent of the relationship between the* `Artist` *and the* `birth_date`*, is the graph shown in Figure 5.8(b) and the mapping between an XML tree and an RDF/S path, will be*[3]*:*

his_biography(x,y),birth_date(y,z) :- {*//Artist*}(*k*),{*./@name*}(*k,x*),
{*./birthdate*}(*k,z*) | *concat(a, x, "_")*,
concat(y, a, z).□

By taking into account the rules that we considered in Section 5.1 we can get a first view on the correspondences between the two schemas presented earlier. Figure 5.9 depicts graphically such correspondences, and the use of different line styles discerns the different cases. The true mappings will be discussed in detail in Section 5.3.2.

[3]In order to identify the instances of Biography (since it has no correspondent in the XML source) we construct an id using the following formula: Artist_id+"_"+birth_date.

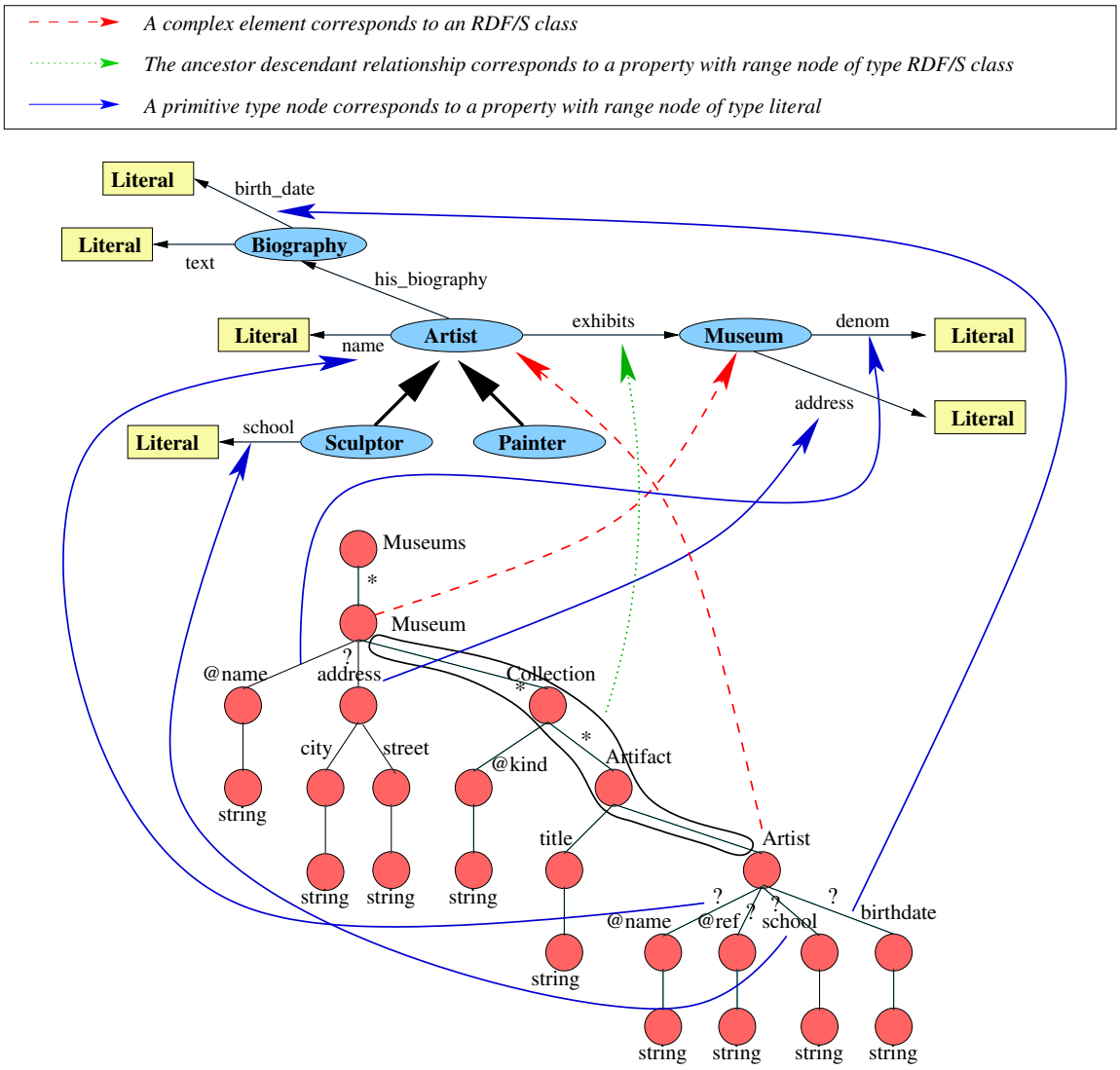


Figure 5.9: The correspondences between the XML and RDF/S schema of our example

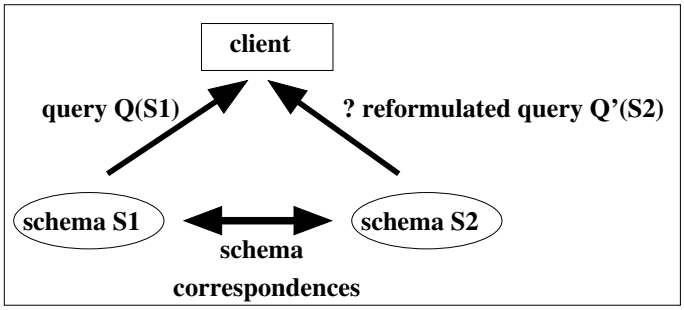


Figure 5.10: The general problem of query reformulation

5.3 Query Reformulation Algorithm

Definition 5.3 *The problem of query reformulation amounts to finding (whenever it exists) a query (or queries) $Q'(S_2)$ over schema S_2 that return(s) the same answer to a given query $Q(S_1)$ over a schema S_1.* □

Query reformulation is graphically depicted in Figure 5.10. The *soundness* of a query reformulation algorithm is achieved when it can find **only** the queries $Q'(S_2)$ that are correct (they return the same results as the query $Q(S_1)$). The *completeness* is achieved when it can find **all** the correct queries $Q'(S_2)$.

5.3.1 Chase & Backchase Algorithms

Our query reformulation algorithm, as will be discussed in Section 5.3.2, uses the Chase & Backchase Algorithm.

Figure 5.11 depicts a general overview of the Chase/Backchase algorithm. More precisely, given (i) the two schemas G (public schema) and S (local schema), (ii) a set C of constraints over G (or over G+S) and (iii) a query Q(G) over G, we can underline the following: Chasing the query Q(G) under constraints C will result in a universal plan U(G+S). Backchasing will examine a set of subqueries from which, only a small fragment Q'(S) (if posed to schema S) is equivalent to Q(G).

Chase [DPT99] consists of a number of *chase steps*. In each chase step, starting with a query Q, we apply all the constraints that can be applied.

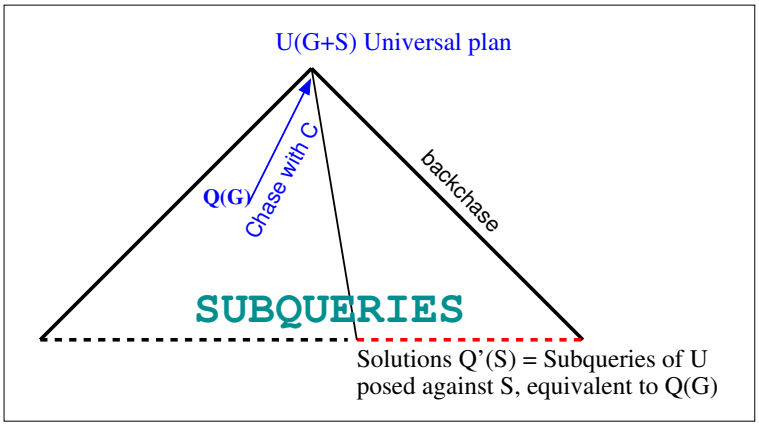


Figure 5.11: General C&B Algorithm

So, given a query $Q : \langle\ A\ \rangle$ and a constraint which assumes that whenever data satisfies A it satisfies B too ($A\ \Rightarrow\ B$), the chase step is:

$$Q : \langle\ A\ \rangle\ \rightsquigarrow\ Q_i : \langle\ A, B\ \rangle$$

The query is chased until no more chase steps apply, and the produced query is called *universal plan*.

Example 5.7 *A simple example that illustrates the basic idea behind chase can be seen in Figure 5.12, where there is a query, a set of mappings and two chase steps. Given the Definition 5.2 these mappings can be interpreted as constraints (two constraints per mapping). Thus, in Figure 5.12 these constraints are also illustrated.* □

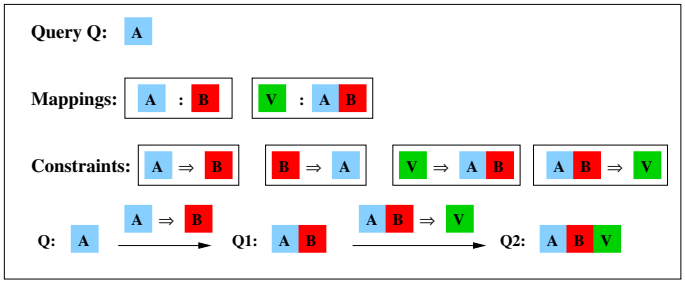


Figure 5.12: Chase example

As stated in [AHV95] the chase of any conjunctive query with any set of embedded dependencies is not guaranteed to terminate. Therefore we need appropriate restrictions, which will ensure the termination of the chase.

[Deu02] states that the chase algorithm in presence of embedded dependencies terminates if the *stratified-witness* property is satisfied. In order to check the satisfaction of this property we use a *chase flow graph*.

Definition 5.4 *Given a set C of constraints, its chase flow graph $G = (V, E)$ is a directed, edge-labeled graph, whose labels can be either $\forall$ or $\exists$. G is constructed as follows: for every relation R of arity α mentioned in C, V contains a node R^i $(1 \leq i \leq \alpha)$. For every pair of relations R, R' of arities α, α', and every constraint*

$$\forall \vec{x}[\ldots \wedge R(u_1, \ldots, u_\alpha) \wedge \ldots \rightarrow R'(v_1, \ldots, v_{\alpha'}) \ldots]$$

in C, E contains the edges $(R_i, R'_j)_{1 \leq i \leq \alpha, 1 \leq j \leq \alpha'}$. Also, whenever the equality $x = y$ appears in the conclusion of the implication, and x, y appear as the i, j-th component of R, R' respectively, E contains the edge (R_i, R'_j). Moreover, if for some j the variable v_j is existentially quantified, the edges $(R_i, R'_j)_{1 \leq i \leq \alpha}$ are labeled with $\exists$, otherwise they are labeled with $\forall$ ([Deu02]). $\square$

We say that a set of constraints has stratified-witness if none of the cycles in its chase flow graph contains an $\exists$-labeled edge.

Example 5.8 *Given the set of constraints $\{\forall x, y\ C(x, y) \rightarrow A(x, y),\ \forall x, y\ A(x, y) \rightarrow \exists z\ B(y, z)\}$ we can construct the chase flow graph shown in Figure 5.13. With the help of this graph, we can observe that the two constraints satisfy the stratified-witness property, since there is no cycle that contains an $\exists$-labeled edge.* $\square$

The existence of a cycle with an $\exists$-labeled edge will prohibit the termination of chase algorithm, since the addition of a new predicate (the predicate with a new existential quantified variable) may always cause the addition of new predicates.

In case of a set C of disjunctive embedded dependencies (DEDs) (instead of embedded dependencies (EDs)) we can think of each $d_i \in C$ as a union of n_i EDs (d_{ij}). Then we have that [Ser05]:

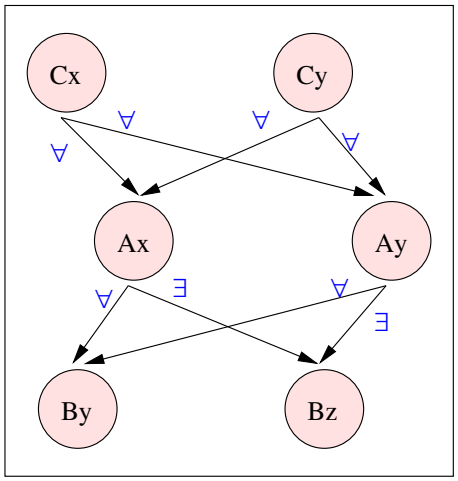


Figure 5.13: Chase flow graph

Proposition 5.1 *The chase of a conjunctive query Q with a set C of DEDs terminates iff for all combinations of m $d_{ij}s$, one for each i, stratified-witness is preserved.* ■

Backchasing [DPT01] is used in order to minimize the query produced during chasing. This is achieved through the inspection of every subquery of the chase result. Each subquery is checked for equivalence to the universal plan, by using (again) the chase algorithm.

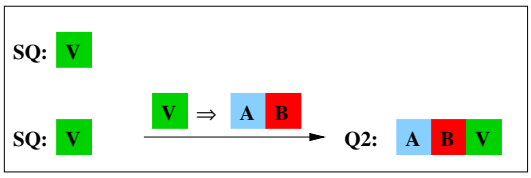


Figure 5.14: Backchase example

Example 5.9 *Given the Example 5.7 the backchase algorithm will conclude to the fact that SQ : V is a minimal reformulation of Q, since SQ (after chasing) is equivalent to Q. Backchase is illustrated in Figure 5.14.*

(We should remind that as discussed in Section 5.2.1 and depicted in Figure 5.12, each mapping between V and $\langle A, B\rangle$ can be interpreted as two constraints ($V \Rightarrow AB$ and $AB \Rightarrow V$).) □

According to [Deu02] the backchase algorithm is sound and complete. This means that it finds minimal subqueries and it finds all of them (when chase algorithm terminates).

Theorem 5.3 *Let Q be a union of conjunctive queries and C a set of DEDs. The chase and backchase algorithms discover all minimal subqueries iff $Q's$ chase with C terminates.* ■

5.3.2 SWIM's Query Reformulation Algorithm

We assume the following steps for query reformulation:

(1) The RQL query posed by the user consists of several RQL patterns (property or class patterns). These patterns are translated into a SWIM logic framework (SWLF) conjunctive query by following specific rules. This translation [Ser05] is based on a set of rules presented thoroughly in Appendix A.

(2) The next step in query reformulation is *chasing* [DPT99]. SWLF query, produced during the first step, is chased in order to be composed with the RDF/S schema information and constraints.

(3) The next step in query reformulation is *backchase*. The universal plan produced in the previous step is backchased in order to be minimized in terms of the view clauses.

(4) In this step the chasing algorithm is used once more in order to compose the minimized SWLF query (query after backchasing) with the defined mappings. For each one of the predicates included in the minimized query we add its mapping rule. In the case when there are two or more mapping rules for a predicate the result is a union of chased queries.

(5) If there are additional constraints from the data sources then these constraints can be used for further minimization of the query produced in the previous step. This step is discussed in details in Section 5.4.

(6) The final SWLF reformulated query is translated into XPath and/or XQuery in order to be executed in the XML data sources (Section 5.5).

SWIM's query reformulation algorithm is sound and complete since it uses the Chase & Backchase algorithm. In other words, (as discussed in the beginning of Section 5.3) it finds all correct queries, which if addressed to XML sources will return the same results as the initial RQL query expressed in terms of the mediated RDF/S schema.

Example 5.10 *The above steps for query reformulation will be detailed through the following example, where the reformulation is done step-by-step. Some steps are simplified in order to make it easier for the reader to follow the sequence of the steps.*

Let's examine the XML source and the virtual RDF/S schema presented in Example 5.6. Additionally, let's suppose that this XML tree corresponds to an XML document with the following content.

```
<Museums>
  <Museum name=‘‘Louvre’’>
     <Collection kind=‘‘Sculpture’’>
        <Artifact>
          <title>Equestrian Statue</title>
          <Artist name=‘‘Leonardo Da Vinci’’/>
        </Artifact>
        <Artifact>
           <title>Madonna with two angels</title>
           <Artist name=‘‘Giovanni di Agostino’’/>
        </Artifact>
     </Collection>
     <Collection kind=‘‘Painting’’>
        <Artifact>
          <title>La Joconde</title>
          <Artist ref=‘‘#Leonardo Da Vinci’’/>
```

```
        </Artifact>
      </Collection>
  </Museum>
...
</Museums>
```

The RDF/S schema of Figure 5.4 is translated in terms of first-order logic, presented in Section 4.3.1, and gives the following instances for the tables CLASS, PROP, C_SUB, P_SUB. The contents of these tables are presented as simple Datalog facts.

CLASS(Artist).
CLASS(Painter).
CLASS(Sculptor).
CLASS(Museum).
CLASS(Biography).
C_SUB(Painter, Artist).
C_SUB(Sculptor, Artist).
PROP(Artist, exhibits, Museum).
PROP(Artist, his_biography, Biography).
PROP(Artist, name, String).
PROP(Biography, birth_date, String).
PROP(Biography, text, String).
PROP(Museum, denom, String).
PROP(Museum, address, String).
PROP(Sculptor, school, String).

Table 5.3: The facts that arise from RDF/S schema

Given this RDF/S schema and the XML grammar we can specify a set of mappings presented in Table 5.4.

Artist(x)	:-	*{//Artist/@name}(x).*
Sculptor(x)	:-	*{//Collection}(k), {./@kind}(k,s),*
		{.//Artist/@name}(k,x), s="Sculpture".
Painter(x)	:-	*{//Collection}(k), {./@kind}(k,s),*
		{.//Artist/@name}(k,x), s="Painting".
Museum(x)	:-	*{//Museum}(k), {./@name}(k,x).*
exhibits(x,y)	:-	*{//Museum}(k), {./@name}(k,y),*
		{.//Artist/@name}(k,x).
exhibits(x,y)	:-	*{//Museum}(k), {./@name}(k,y),*
		{.//Artist/@ref}(k,x).
denom(x,y)	:-	*{//Museum}(m), {./@name}(m,y), x=y.*
address(x,y)	:-	*{//Museum}(m), {./@name}(m,x),*
		{./address}(m,b),{./city}(b,s),
		{./street}(b,q)| concat(y,s,q).
his_biography(x,y),birth_date(y,z)	:-	*{//Artist}(k),{./@name}(k,x),*
		{./birthdate}(k,z) | concat(a, x, "_"),
		concat(y, a, z).
name(x,y)	:-	*{//Artist}(m), {./@name}(m,y), x=y.*
school(x,y)	:-	*{//Collection}(k), {./@kind}(k,s),*
		{.//Artist}(k,m), {./@name}(m,x),
		{./school}(m,y), s="Sculpture".

Table 5.4: The mappings between XML and RDF/S

Let's suppose that a user poses an RQL query to SWIM. This RQL query is expressed in terms of the RDF/S schema of Figure 5.4, and returns the artists that their creations are exhibited in Louvre.

SELECT X
FROM {X}exhibits{Y}, {Y}denom{Z}
WHERE Z = "Louvre"

Step 1 *The first step of the query reformulation includes the translation of the above RQL query in terms of the SWIM logic framework predicates. The result of this translation is a rewriting of the query in terms of the SWIM internal framework:*

ans(x) :- P_SUB(A, exhibits), P_EXT(x, A, y),
P_SUB(B, denom), P_EXT(y, B, z),
z="Louvre".

Step 2 *The following step is the application of chase algorithm. Given the facts presented earlier (Table 5.3) the query is transformed to:*

ans(x) :- P_SUB(A, exhibits), P_EXT(x, A, y),
P_SUB(B, denom), P_EXT(y, B, z),
A = exhibits, B = denom, z ="Louvre".

In Section 4.3.2 we defined some constraints that are used for capturing the RDF/S semantics. Thinking of these constraints we can point out that some of them can be applied. There is a constraint that declares that every statement in the extent of a property implies the existence of the corresponding property (RDF/S Basic Constraint (2)):

$$\forall x, p, y[P_EXT(x,\ p,\ y) \rightarrow \exists c, dPROP(c,\ p,\ d)]$$

can be applied in the query above and the result is:

ans(x) :- P_EXT(x, A, y), P_EXT(y, B, z),
PROP(s, A, t), PROP(m, B, k),
A = exhibits, B = denom, z = "Louvre".

We would like to point out that from the above query have been removed the predicates that are not useful anymore (we used them and we are not going to use them again). The reason for this elimination is only to improve the readability of the reformulated queries. So, predicates P_SUB(A, exhibits) and P_SUB(B, denom) have been removed. Further using of the above facts (Table 5.3) will lead to the following query:

ans(x)	:-	*P_EXT(x, A, y), P_EXT(y, B, z),*
		PROP(s, A, t), PROP(m, B, k),
		A = exhibits, B = denom,
		s = Artist, t = Museum,
		m = Museum, k = String, z = "Louvre".

Now another constraint can be applied. This is the constraint which states that in a valid RDF description base the subject/object resources in every statement are instances of the property's domain/range classes (either direct or a subclass' instances) (RDF/S Basic Constraint (12)):

$$\forall a, p, b, x, y[[PROP(a,\ p,\ b) \wedge P_EXT(x,\ p,\ y)] \rightarrow$$
$$\exists c, d[C_SUB(c,\ a) \wedge C_SUB(d,\ b) \wedge C_EXT(c,\ x) \wedge C_EXT(d,\ y)]]$$

and we conclude to the following query:

ans(x)	:-	*P_EXT(x, A, y), P_EXT(y, B, z),*
		PROP(s, A, t), PROP(m, B, k),
		C_SUB(a, s), C_EXT(a, x), C_SUB(b, t), C_EXT(b, y),
		C_SUB(c, m), C_EXT(c, y), C_SUB(d, k), C_EXT(d, z),
		A = exhibits, B = denom,
		s = Artist, t = Museum,
		m = Museum, k = String, z = "Louvre".

Further using of RDF/S facts can change our query to:

ans(x) :- *P_EXT(x, A, y), P_EXT(y, B, z),*
PROP(s, A, t), PROP(m, B, k),
C_SUB(a, s), C_EXT(a, x), C_SUB(b, t), C_EXT(b, y),
C_SUB(c, m), C_EXT(c, y), C_SUB(d, k), C_EXT(d, z),
A = exhibits, B = denom,
s = Artist, t = Museum, b = Museum,
m = Museum, c = Museum, k = String, d = String,
z = "Louvre".

Removing some of the redundant predicates again we get:

ans(x) :- *P_EXT(x, A, y), P_EXT(y, B, z),*
C_SUB(a, s), C_EXT(a, x), C_EXT(b, y),
C_EXT(c, y), C_EXT(d, z),
A = exhibits, B = denom,
s = Artist, b = Museum,
c = Museum, d = String, z = "Louvre".

In the previous query we haven't considered the subclasses of `Artist`*. The addition of* `Artist's` *subclasses will lead to a union of queries and we prefered to show it in an extra step in order to be more clear. So:*

ans(x) :- *P_EXT(x, A, y), P_EXT(y, B, z),*
C_EXT(a, x), C_EXT(b, y),
C_EXT(c, y), C_EXT(d, z),
A = exhibits, B = denom,
a = Artist, b = Museum,
c = Museum, d = String, z = "Louvre".

$\bigcup$

ans(x) :- *P_EXT(x, A, y), P_EXT(y, B, z),*
C_EXT(a, x), C_EXT(b, y),

C_EXT(c, y), C_EXT(d, z),
A = exhibits, B = denom,
a = Painter, b = Museum,
c = Museum, d = String, z = "Louvre".

$\bigcup$

ans(x) :- P_EXT(x, A, y), P_EXT(y, B, z),
C_EXT(a, x), C_EXT(b, y),
C_EXT(c, y), C_EXT(d, z),
A = exhibits, B = denom,
a = Sculptor, b = Museum,
c = Museum, d = String, z = "Louvre".

Step 3 *In this step the previous query is backchased in order to be minimized in terms of the view clauses. Of course some steps of the backchase (the elimination of some predicates) were nested to the above process (for simplicity reasons). Further backchasing will eliminate some of the redundant predicates and conclude to the union of the following queries:*

ans(x) :- P_EXT(x, A, y), P_EXT(y, B, z),
C_EXT(a, x), C_EXT(b, y),
A = exhibits, B = denom,
a = Artist, b = Museum,
z = "Louvre".

$\bigcup$

ans(x) :- P_EXT(x, A, y), P_EXT(y, B, z),
C_EXT(a, x), C_EXT(b, y),
A = exhibits, B = denom,
a = Painter, b = Museum,

z = *"Louvre"*.

$\bigcup$

ans(x) :- *P_EXT(x, A, y), P_EXT(y, B, z),*
C_EXT(a, x), C_EXT(b, y),
A = *exhibits*, B = *denom*,
a = *Sculptor*, b = *Museum*,
z = *"Louvre"*.

What we should indicate is that the query above can be translated into a query that includes the view clauses of the RDF/S schema. This translation is trivial:

ans(x) :- *A(x, y), B(y, z),*
a(x), b(y),
A = *exhibits*, B = *denom*,
a = *Artist*, b = *Museum*,
z = *"Louvre"*.

$\bigcup$

ans(x) :- *A(x, y), B(y, z),*
a(x), b(y),
A = *exhibits*, B = *denom*,
a = *Painter*, b = *Museum*,
z = *"Louvre"*.

$\bigcup$

ans(x) :- *A(x, y), B(y, z),*
a(x), b(y),
A = *exhibits*, B = *denom*,
a = *Sculptor*, b = *Museum*,
z = *"Louvre"*.

Step 4 *The following step in query reformulation includes further chasing. This is done in order to compose the minimized query with the mappings presented above. The result (having removed the view clauses) will be:*

---

ans(x) :- {*//Museum*}(k), {*./@name*}(k,y), {*.//Artist/@name*}(k,x),
{*//Museum*}(m), {*./@name*}(m,z), $y = z$,
{*//Artist/@name*}(x),
{*//Museum*}(q), {*./@name*}(q,y),
$z =$ *"Louvre"*.

$\bigcup$

ans(x) :- {*//Museum*}(k), {*./@name*}(k,y), {*.//Artist/@ref*}(k,x),
{*//Museum*}(m), {*./@name*}(m,z), $y = z$,
{*//Artist/@name*}(x),
{*//Museum*}(q), {*./@name*}(q,y),
$z =$ *"Louvre"*.

$\bigcup$

ans(x) :- {*//Museum*}(k), {*./@name*}(k,y), {*.//Artist/@name*}(k,x),
{*//Museum*}(m), {*./@name*}(m,z), $y = z$,
{*//Collection*}(p),{*./@kind*}(p,w),{*.//Artist/@name*}(p,x),w=*"Painting"*,
{*//Museum*}(q), {*./@name*}(q,y),
$z =$ *"Louvre"*.

$\bigcup$

ans(x) :- {*//Museum*}(k), {*./@name*}(k,y), {*.//Artist/@ref*}(k,x),
{*//Museum*}(m), {*./@name*}(m,z), $y = z$,
{*//Collection*}(p),{*./@kind*}(p,w),{*.//Artist/@name*}(p,x),w=*"Painting"*,
{*//Museum*}(q), {*./@name*}(q,y),
$z =$ *"Louvre"*.

$\bigcup$

ans(x) :- {*//Museum*}(k), {*./@name*}(k,y), {*.//Artist/@name*}(k,x),

$\{//Museum\}(m),\ \{./@name\}(m,z),\ y = z,$
$\{//Collection\}(p),\{./@kind\}(p,w),\{.//Artist/@name\}(p,x),w=\text{“Sculpture”},$
$\{//Museum\}(q),\ \{./@name\}(q,y),$
$z = \text{“Louvre”}.$

$\bigcup$

$ans(x)$:- $\{//Museum\}(k),\ \{./@name\}(k,y),\ \{.//Artist/@ref\}(k,x),$
$\{//Museum\}(m),\ \{./@name\}(m,z),\ y = z,$
$\{//Collection\}(p),\{./@kind\}(p,w),\{.//Artist/@name\}(p,x),w=\text{“Sculpture”},$
$\{//Museum\}(q),\ \{./@name\}(q,y),$
$z = \text{“Louvre”}.$

□

5.4 Improving Reformulations with XML Data Semantics

Until now the RQL query has been reformulated with the use of general constraints capturing the semantics of both the RDF/S and XML data models. However, the use of additional constraints from the XML data sources will improve query reformulation. XML, for example, supports the specification of keys and keyrefs, as well as value enumerations for attributes. Such additional constraints can improve our reformulations by eliminating redundant queries or by simplifying the generated queries (removing some of their predicates). Being able to use such additional constraints is very important and this will be clarified in the following example (it is an extension of the previous one).

Example 5.11 *The DTD presented before can inform us about the attribute* `kind` *of each* `Collection`*. Its definition declares that the attribute* `kind` *is an enumerated*

type, whose value can be either "Sculpture" or "Painting". This can be expressed through the following constraint:

$$\forall\ x1, x2\ [XMLroot(x1), D(x1,\ x2), T(x2,\ \text{“}Collection\text{”}) \rightarrow$$
$$\exists\ x3, v\ [C(x2,\ x3),\ A(x3,\ \text{“}kind\text{”},\ v),\ v = \text{“}Painting\text{”}] \vee$$
$$\exists\ x3, v\ [C(x2,\ x3),\ A(x3,\ \text{“}kind\text{”},\ v),\ v = \text{“}Sculpture\text{”}]]$$

Step 5 *Using this constraint we can conclude to the following query:*

ans(x) :- {//Museum}(k), {./@name}(k,y), {.//Artist/@name}(k,x),
{//Museum}(m), {./@name}(m,z), y = z,
{//Artist/@name}(x),
{//Museum}(q), {./@name}(q,y),
z = "Louvre".

$\bigcup$

ans(x) :- {//Museum}(k), {./@name}(k,y), {.//Artist/@ref}(k,x),
{//Museum}(m), {./@name}(m,z), y = z,
{//Artist/@name}(x),
{//Museum}(q), {./@name}(q,y),
z = "Louvre".

The first query comprises a superset of the second one, since the second is a restriction of the first. Taking into account that we have a union of queries we can eliminate the second query and the result is:

ans(x) :- {//Museum}(k), {./@name}(k,y), {.//Artist/@name}(k,x),
{//Museum}(m), {./@name}(m,z), y = z,
{//Artist/@name}(x),
{//Museum}(q), {./@name}(q,y),
z = "Louvre".

Now let's assume that the grammar for the XML document defined previously is not given through a DTD but through an XML Schema. XML Schema is able

to define keys and keyrefs (foreign keys). In particular it has been declared that the attribute **name** *is a key for the* **Museum**, *while the attribute* **name** *is a key for the* **Artist**.

In order to express the fact that an element (or attribute) e.g., the **name** *consists a key of another element e.g. of the* **Museum** *we need the following three constraints.*

- *Every Museum has a name*

$$\forall\ x1, x2\ [XMLroot(x1), D(x1,\ x2), T(x2,\ \text{“}Museum\text{”}) \rightarrow \exists\ x3, v\ C(x2,\ x3), A(x3,\ \text{“}name\text{”},\ v)]$$

- *For every Museum having two “name”s these “name”s are equal*

$$\forall\ x1, x2, x3, v1, x4, v2\ [XMLroot(x1), D(x1,\ x2), T(x2,\ \text{“}Museum\text{”}), C(x2,\ x3), C(x2,\ x4), A(x3,\ \text{“}name\text{”},\ v1), A(x4,\ \text{“}name\text{”},\ v2) \rightarrow\ v1 = v2]$$

- *For every two Museums having the same “name” these Museums are the same*

$$\forall\ x1, x2, x3, x4, x5, v1, v2\ [XMLroot(x1), D(x1,\ x2), T(x2,\ \text{“}Museum\text{”}), D(x1,\ x3), T(x3,\ \text{“}Museum\text{”}), C(x2,\ x4), C(x3,\ x5), A(x4,\ \text{“}name\text{”},\ v1), A(x5,\ \text{“}name\text{”},\ v2), v1 = v2\ \rightarrow\ x2 = x3]$$

Similarly, the key of the **Artist** *(attribute* **name***) is expressed as:*

- *Every Artist has a name*

$$\forall\ x1, x2\ [XMLroot(x1), D(x1,\ x2), T(x2,\ \text{“}Artist\text{”}) \rightarrow \exists\ x3, v\ C(x2,\ x3), A(x3,\ \text{“}name\text{”},\ v)]$$

- *For every Artist having two “name”s these “name”s are equal*

$$\forall\ x1, x2, x3, v1, x4, v2\ [XMLroot(x1), D(x1,\ x2), T(x2,\ \text{“}Artist\text{”}), C(x2,\ x3), C(x2,\ x4), A(x3,\ \text{“}name\text{”},\ v1), A(x4,\ \text{“}name\text{”},\ v2) \rightarrow\ v1 = v2]$$

- *For every two Artists having the same "name" these Artists are the same*

$$\forall\ x1, x2, x3, x4, x5, v1, v2\ [XMLroot(x1), D(x1,\ x2), T(x2,\ \text{"}Artist\text{"}),$$
$$D(x1,\ x3), T(x3,\ \text{"}Artist\text{"}), C(x2,\ x4), C(x3,\ x5),$$
$$A(x4,\ \text{"}name\text{"},\ v1), A(x5,\ \text{"}name\text{"},\ v2), v1 = v2\ \rightarrow\ x2 = x3]$$

Additionally, XML Schema could declare that `Artist's` *attribute* `ref` *is a keyref and it refers to the key of another* `Artist`*.*

The constraint that expresses the keyref of the `Artist's` *attribute* `ref` *to the* `Artist's` *attribute* `name` *is the following:*

- *For every Artist having an attribute "ref" there is an Artist, which has an attribute "name", such, that the value of "ref" is equal to the value of "name"*

$$\forall\ x1, x2, x3, a\ [XMLroot(x1), D(x1,\ x2), T(x2,\ \text{"}Artist\text{"}), C(x2,\ x3),$$
$$A(x3,\ \text{"}ref\text{"},\ a)\ \rightarrow\ \exists\ x4, x5, v\ D(x1,\ x4)$$
$$T(x4,\ \text{"}Artist\text{"}), C(x4,\ x5), A(x5,\ \text{"}name\text{"},\ v), v = a]$$

Let's suppose that we have the two queries in which we concluded above (before discovering that the first one is a superset of the second and the elimination of the second). Then, the second query that appears in the union includes a reference that, as defined above, is a keyref. This reference (`@ref` *as a child of an* `Artist`*) will cause the addition of the predicates:*

$$\{//Artist/@name\}(o),\ o = x$$

This would lead to different reformulation steps and probably would cause other simplifications of the queries. However, in this particular case (by eliminating the second query) we cannot apply this constraint.

In the following reformulation steps, the produced query has many (more than two) `Museums` *that have the same* `name` *and two* `Artists` *with the same attribute*

`name`. Since the attributes `name` are the keys of the `Museum` and `Artist` respectively, we can further reformulate the query by applying the key constraints. In fact, the result includes all the predicates and what we really add is some equalities between them. However, in order to make the advantage of using these constraints clearer, we prefer to show only the necessary, for further reformulation, predicates. The result of applying the key constraints is:

ans(x)	*:-*	*{//Museum}(s), {./@name}(s,y),*
		{.//Artist/@name}(s,x), z=y, z="Louvre".

The "true" (SWLF) representation of this query is[4]*:*

ans(x)	*:-*	*XMLroot(x0), D(x0, x1), E(x1, x2, "Museum"),*
		C(x2, x3), A(x3, "name", y),
		D(x2, x4), E(x4, x5, "Artist"), C(x5, x6), A(x6, "name", x),
		y = "Louvre".

□

Looking at the result of the example above it is obvious that the first step towards query reformulation has been completed. The initial RQL query has been transformed to a query that is expressed in terms of the local XML sources. However, this query cannot be executed in an XML source.

5.5 Query Translation

As illustrated above, the effort of the query reformulation algorithm is to end up to the minimal possible meaningful queries. However, no matter what the number and the complexity of these queries are, they are not capable of returning results if addressed to XML sources. So, there is a need for translating these queries to query languages appropriate for retrieving data from the XML sources. There are

[4]Reminder: The interpretation of a//b (according to the semantics of XPath2.0) is: a/descendant-or-self::node/child::b.

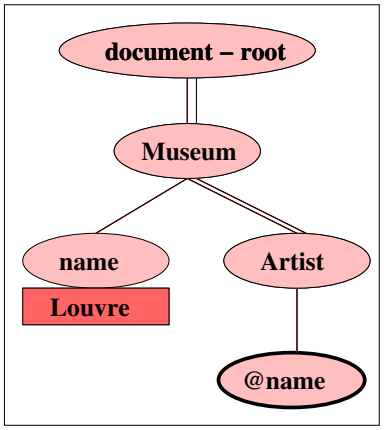


Figure 5.15: The reconstruction of the reformulated query

two target XML query languages that have to be examined: XPath and XQuery. The issues involved in the translation of reformulated queries into XPath and XQuery are discussed below.

The query (or queries) in which the reformulation concludes is expressed in terms of the SWLF representation of the XML presented in Section 4.2.1 (i.e., it has element nodes, attribute nodes, child or descendant relationships and one root). Processing these terms can lead to the reconstruction of a tree which represents the query.

In this step of query reformulation we handle the built-in predicates that were used in cases of more complex mappings. These predicates were not handled during the previous steps of reformulation and they are manipulated now according to the capabilities of the target languages.

Example 5.12 *The query in which we conclude to in Example 5.11, leads to the reconstruction of the tree shown in Figure 5.15.*

As explained in Example 4.1 the nodes that are connected through a line have the parent-child relationship, while double line stands for ancestor-descendant relationship. The node having a thicker outline is the one whose content is the returned query value.□

The desired result of such a translation is to get an RDF description. SWIM users pose RQL queries and they expect to get RDF results. So, it is important to get such results with as few transformations as possible.

5.5.1 The XPath Translation

XPath, as explained in Section 2.1.2, is a non-XML language used for navigation in the tree that represents an XML document. The result of an XPath expression may be a selection of nodes from the input documents, or an atomic value, or more generally, any sequence allowed by the XML data model.

Example 5.13

Step 6 *The XPath translation of the reformulated query (based on the tree representation presented in Example 5.12) is:*

//Museum[./@name = "Louvre"]//Artist/@name

and given the XML data illustrated in Example 5.10 we get the following result:

<name>Giovanni di Agostino</name>
<name>Leonardo Da Vinci</name>

□

This set of nodes, returned by an XPath expression, is not an XML document. XPath does not have the ability to return a well-formed XML document and of course not an RDF description (as desired). So, the results above should be transformed (with a transformation function that receives nodes) in order to construct an RDF description.

As shown in Section 2.1.2 there are many differences between XPath 1.0 and its successor XPath 2.0. These differences are taken into consideration in the transformation of the reformulated query.

Both XPath 1.0 and 2.0 are able to express most of the required features such as navigation through the tree that represents an XML document, but they are not able to project over two node values. If in the Example 5.12 we wanted to return both the `name` of the `Artist` and the `denomination` of the `Museum` these XPath versions could not offer such option.

XPath 1.0 does not support explicit quantification i.e., to concatenate the `first` and `last name` of each `Artist`. On the contrary, XPath 2.0 is able to express more complex queries due to the fact that we can use for, let, etc. and variable bindings. However, the use of such features would lead in complex XPath expressions.

Finally, we have chosen to use XPath 1.0, since we prefer to use simple XPath expressions, and according to its restrictions we use it only for *simple RQL queries* (queries that project on one variable) and *simple XML constraints* (no complex keys).

5.5.2 The XQuery Translation

The major advantage of XQuery against XPath is that the former is able to construct elements and produce a well-formed XML document. With appropriate tag selection in the return clause we can create a valid RDF description according to the employed RDF/S schema. For the translation we use XQuery 1.0.

Example 5.14

Step 6 *The XQuery translation of the reformulated query (based on the tree representation of Example 5.12) is:*

```
<RDF>
{
  <Bag>
  {
    for $var0 in document("art.xml")
    for $var1 in $var0//Museum
    for $var2 in $var1/@name
    for $var3 in $var1//Artist
    for $var4 in $var3/@name
    where ($var2/text()="Louvre")
    return
      <li>$var4/text()</li>
```

```
    }
    </Bag>
  }
</RDF>
```

and given the XML data shown in Example 5.10 we get the following result:

```
<RDF>
    <Bag>
        <li>Giovanni di Agostino</li>
        <li>Leonardo Da Vinci</li>
    </Bag>
</RDF>
```

□

5.6 How SWIM Addresses Heterogeneity Issues

As discussed in Chapter 3 heterogeneity is one of the most complicated issues that should be taken into consideration in a data integration system. In this chapter, we consider a classification of heterogeneity problems and we emphasize the way in which SWIM can efficiently solve them.

Definition 5.5 *Heterogeneity refers to the degree of dissimilarity between the component data sources that make up a data integration system.* □

Heterogeneity occurs at different integration levels. On a lower level, heterogeneity comes from different *hardware platforms* or different *operating systems* and *communication protocols*. On a higher level, heterogeneity comes from different *understanding* and *modeling* of the same concept.

Logical heterogeneity arises when different people represent the same concept differently, and this is the reason that it cannot be resolved easily. Logical heterogeneity

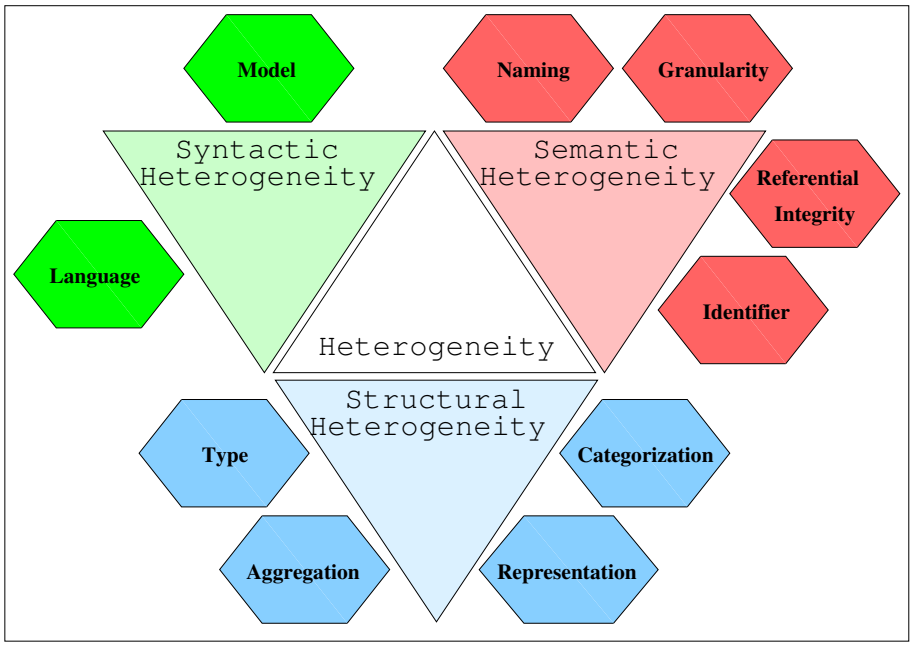


Figure 5.16: Heterogeneity map

can be further divided in three types of dissimilarities: *syntactical*, *structural* and *semantical* [SK92] [KS91] [HST99]. These axes of heterogeneity are discussed below through the differences that may arise between the XML and the RDF, which is our application scenario.

The categories of heterogeneity discussed below are presented in the Figure 5.16.

5.6.1 Syntactic Heterogeneity

Definition 5.6 *Syntactic heterogeneity refers to the encoding of the same concept in different data models (or natural languages).* □

There are two types of syntactic heterogeneity:

- *Language heterogeneity.* It is a simple kind of heterogeneity that refers to different natural languages. For example, the RDF/S schema may contain a class called `Artifact`, while the one of the XML sources uses the Spanish language and has an element that is called `Obra de arte`.

- *Data model heterogeneity.* It refers to the fact that there may exist different

data models that should be integrated. In case of XML and RDF/S this kind of heterogeneity refers to the fact that XML data model is a node-labeled tree, while RDF/S is a node-labeled graph with labels at the edges too. This kind of discrepancies and their solution has been discussed thoroughly in Section 5.1.

5.6.2 Structural Heterogeneity

Definition 5.7 *Structural heterogeneity arises when the same concept is represented by different structures.* □

- *Categorization conflicts*. This kind of heterogeneity arises due to the fact that some languages (like RDF/S) are able to define subsumption hierarchies (for classes or properties), while others are not (XML).

  Example 5.15 *Let's consider again the Example 5.10. In the RDF/S schema there are two subclasses of class* `Artist`, `Painter` *and* `Sculptor`. *However, XML declares that there are elements called* `Artist` *and they are characterized as painters or sculptors according to the value of the attribute* `kind` *of the* `Collection` *in which they exhibit their creations.*

  The solution that can be given with the help of our mappings is:

  $$\begin{aligned} \textit{Painter(x)} &:- \{\textit{//Collection}\}(a),\ \{\textit{.//Artist/@name}\}(a,x),\\ &\quad\ \{\textit{./@kind}\}(a,s),\ s=\text{“Painting”}.\\ \textit{Sculptor(x)} &:- \{\textit{//Collection}\}(a),\ \{\textit{.//Artist/@name}\}(a,x),\\ &\quad\ \{\textit{./@kind}\}(a,s),\ s=\text{“Sculpture”}.\\ \textit{Artist(x)} &:- \{\textit{//Artist/@name}\}(x). \end{aligned}$$

  □

  This solution is feasible in cases where in XML there is a notion of categorization (even in simple tag alternatives). If all the `artists` (`painters` and `sculptors`) were under the element `Collection`, without any further categorization, the mapping will be:

$$\text{Artist(x)} \quad :- \quad \{//\text{Artist/@name}\}(\text{x}).$$

- *Typing conflicts.* This kind of heterogeneity arises from the fact that the same concept can have different types. For example, the `birthdate` of an `Artist` in the XML document may be an element of type "string". On the other hand, in the RDF description `birth_date` may be a property with range of type "date". Even in this particular case the use of built-in predicates (i.e., concat) will solve the problem, in general, it cannot be solved without loss of information.

  Another aspect of this kind of heterogeneity is the differences concerning ordering. As discussed earlier, XML element nodes are ordered, while in RDF there isn't such option. Since in our RDF/S first-order logic representation, order is disregarded, we cannot provide solution for this kind of typing conflicts.

  One more typing problem occurs in cases that RDF/S represents a concept as a class or property name (typing information), while in XML this information is provided only at data level.

  Example 5.16 *Let's assume that in the RDF description there are* `sculptors` *who* `sculpt sculptures`*. On the other hand, the XML document contains* `Artists`*, (with aid as identification attribute) that have a* `type` *and they create* `Artifacts`*. That is if we want to talk about* `sculptors` *in the XML document we have to talk about* `Artists` *whose* `type` *is* `''Sculptor''`*. This leads to the following mapping:*

  $$\begin{aligned} \textit{Sculptor(x)} \quad :- \quad & \{\textit{//Artist}\}(a),\ \{\textit{./type}\}(a,b), \\ & \{\textit{./@aid}\}(a,x),\ b = \textit{"Sculptor"}. \end{aligned}$$

  □

- *Aggregation conflicts* arise from the fact that some information can be divided into smaller pieces of information. A particular and common example is the name. A person's `name`, in an RDF description, can be of type string, i.e.,

"Leonardo Da Vinci" while in an XML document a person's `name` can be an element with two components: `first_name` ("Leonardo") and `last_name` ("Da Vinci"). Many of these discrepancies can be eliminated with the use of the non interpreted functions (e.g., functions for handling strings) that we are able to use. In this particular example the solution provided by our mappings is:

name(x, y) :- {//Artist}(a), {./@aid}(a,x), {./first_name}(a,s),
{./last_name}(a,t) | concat(y,s,t).

- *Representation conflicts* include occasions of different representation. For instance, in our XML DTD, `Artifacts` contain information about their `Artists`, while our RDF/S schema `Artists` exhibit in a `Museum` and no `Artifact` information is available.

5.6.3 Semantic Heterogeneity

Definition 5.8 *Semantic heterogeneity refers to the fact that data represented in different systems may be subject to different interpretations.* □

The distinction between semantic and structural heterogeneity is not always clear-cut. Structural heterogeneity refers basically to the structure of the data, while semantical refers to the domain of the concepts (their interpretation). However, it is true that sometimes the structure of a piece of information provides the user with at least a part of the meaning of this information.

The structure of the data (in isolation) cannot always lead us to structural heterogeneity resolution. The semantics of the data can give more refined information about what the structure really represents. The basic categories of semantic heterogeneity are:

- *Naming conflicts*, which are a common problem. There are two types of naming conflicts. The first is the case of *synonyms*. Two terms are called synonyms if they have the same meaning. In a data integration system this problem occurs when two terms are used as they have they same meaning. In an RDF

description, for example, we may use the term ‘‘`Technique`’’ for describing the technique used by a painter for the construction of its paintings. On the other hand, XML source uses the term ‘‘`Method`’’ in order to express the same technique.

The second type of naming conflicts is the case of *homonyms*. Homonyms are the terms that can have two or more different meanings. For example, the term ‘‘`Java`’’ can be used in an RDF description for the programming language Java, while an XML source uses the same term in order to talk about the island called Java.

This kind of heterogeneity can be easily solved through mappings, supposing that we can identify that the terms (or the term) of the RDF and XML have the same (or different) meaning(s). In case of homonyms there is prerequisite that the different meanings (of the term) exist in both RDF and XML, i.e., in RDF description there is a concept that corresponds to the Java island. Otherwise, such a mapping does not exist.

- *Granularity conflicts* are further divided into two categories: scaling conflicts and precision conflicts. *Data scaling* dissimilarities appear when equivalent concepts have different units and measures. For example, an RDF description may represent the dimensions of a painting in centimeters (`cm`), while in XML these dimensions are given in inches (`in`). *Precision conflicts* appear when the same concept has different precision. For example, an artist may have created his artifacts during a period which can be expressed through specific reference to years (i.e., Picasso created “Les Demoiselles d’Avignon” in 1907) or through an art period (i.e., Picasso created “Les Demoiselles d’Avignon” during Cubism (1906-1912)).

  Mappings are usually able to bridge these differences, but this process may lead to loss of information.

- *Identifier conflicts* are one of the most intricate heterogeneity problems and concerns the identity of the concepts. RDF resources are identified by a unique

URI. This URI can be specified by the domain expert that determines the RDF description. In the case of anonymous nodes, the domain application expert should declare which resources are identical. This can be accomplished by specifying the identical resources with the use of a constraint (defined in a similar way with the constraints of Section 5.4). This constraint will have the same meaning with the OWL property "owl:sameAs" [OWL]. However, in an XML document there is no such identifier for every element (attribute). Additionally, XML has the ability to identify its elements at different granularity levels. That is an XML element can be unidentified, globally identified or locally identified in a specific scope.

Since URIs are unique they should be mapped to unique XML keys. In particular:

- If there are global keys, mapping is simple (like the cases discussed in previous examples i.e., Example 5.10).
- If there are local keys then they should become global and this can be done by adding as prefix the name of the tag in whose scope the key has been locally defined.

  Example 5.17 *Let's assume that the key* ***''name''*** *of an* ***Artist*** *is defined in the scope of the* ***Artist*** *(local key). That means that every* ***Artist*** *has a different* ***name*** *but it can be the same with the* ***name*** *of a* ***Museum*** *(if it is locally defined too). The definition of the* ***name*** *as a local key (in an XML Schema syntax) will be:*

  <key name="name">
  <selector xpath="//Artist"/>
  <field xpath="@name"/>
  </key>

  and then the mapping will be:

  Artist(x) :- {//Artist}(a), {./@name}(a,b) | concat(x, "Artist_", b).

– If there is no information about keys we have to identify all the elements in the XML document globally. This can be done if we compose a local key for every element (the elements with textual children), by concatenating the values of its children, and make this local key global with the process described above.

The expressiveness of our mappings allows the manipulation of the keys in order to produce unique identifiers.

- *Referential integrity* conflict is based on the fact that RDF/S is able to specify clearly the domain and the range of each property, unlike XML DTDs. Of course, XML does not have properties, but there exist element references (IDREF(S) in DTD/XML Schema) that cannot be checked for their validity. For example, an `Artist` can have a reference to a `Museum` while an instance on an `Artist` can refer to an `Institute` instead of a `Museum`. The mappings that we use are able to enforce those restrictions, i.e.,

$$\begin{aligned} \text{exhibits(x, y)} \quad :- \quad & \{\text{//Artist}\}(a), \{./@\text{aid}\}(a, x), \\ & \{./\text{exhibits}\}(a, k), \{\text{//Museum}\}(y), y = k. \end{aligned}$$

Chapter 6

SWIM Architecture

In this chapter, we detail the architecture of the SWIM middleware that integrates data from heterogeneous sources on the Semantic Web.

The general architecture of SWIM is depicted in Figure 6.1. In this figure we can observe the basic contents of the middleware, which are: (i) the RDF/S virtual schema used for the integration and (ii) the mapping rules that specify the correspondences between this RDF/S schema and the local sources (Datalog). Since this thesis focuses on XML sources, the mappings stored in the middleware concern the correspondences between an RDF/S schema and XML data.

Queries, in SWIM, are formulated against the mediated RDF/S schema using RQL or virtual RDF/S views specified in RVL. The results of the queries that should be returned to the user are valid RDF descriptions. The RQL queries, after the reformulation that they experience, are translated into queries that are appropriate for the XML sources in which they are addressed to. As discussed in Chapter 5, RQL queries are translated into XPath and/or XQuery before being sent to the XML sources. The results of the execution of the queries are sent back to the middleware and afterwards to the user.

This reformulation is done by a module that is responsible for query reformulation. This query reformulation engine is a basic component of the architecture and is discussed in details below.

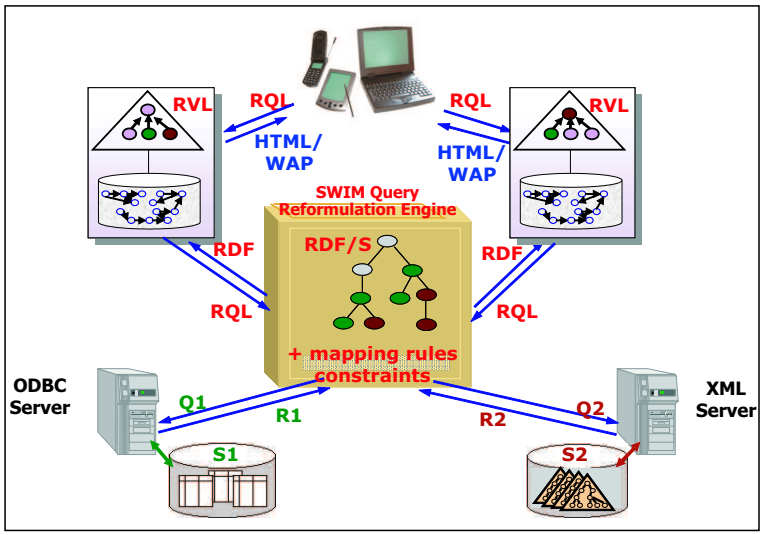


Figure 6.1: The ICS-FORTH SWIM Architecture

6.1 SWIM Query Reformulation Engine

Figure 6.2 depicts query reformulation inside the SWIM server. What is submitted to the query reformulation engine is:

- The RDF/S Schema
- The RDF/S semantics (the basic constraints that capture the semantics of RDF (Section 4.3.2))
- The mappings between XML and RDF/S
- The constraints (if any) coming from the data sources and of course,
- The RQL query posed by the user

In Figure 6.2 we can see the internal information flow of the query reformulation engine. There are four basic components:

- The *SWLF compiler*. The basic task of this component is to translate the RDF/S schema into the SWIM internal logic framework (SWLF) (Chapter 4) and to translate the RQL query into a Datalog format.

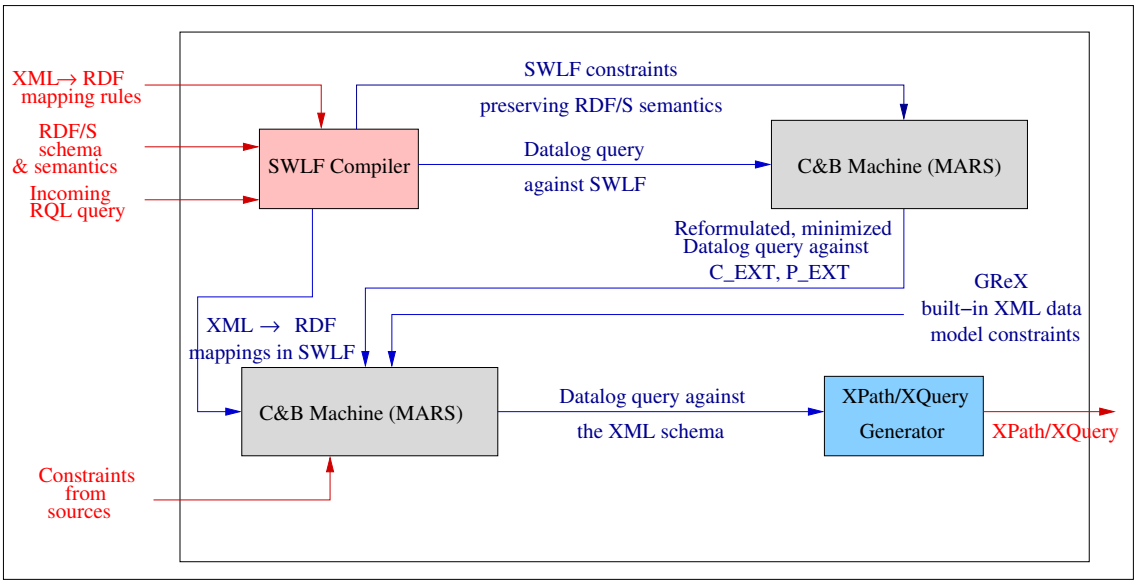


Figure 6.2: Architecture of the SWIM Query Reformulation Engine

- The *first call of the chase/backchase machine* receives the output of the SWLF compiler as input. It is responsible for the first chase and backchase of the query until its minimization and its reformulation against the view clauses of RDF/S (or, equivalently, against P_EXT and C_EXT predicates).

- The output of the first call of chase/backchase machine is further chased in the *second call of chase and backchase machine* with the use of the XML→RDF mappings and of the additional constraints (if any) from the XML sources. Moreover, in this step the GReX relations (the first-order logic XML representation defined in Section 4.2) are used, along with the constraints that capture the XML semantics.

- The *XPath/XQuery generator*. The output of the third step is an optimized query expressed in terms of the XML documents. This output is further processed by this module that is responsible for the construction of the XQuery and/or XPath expressions, which consist the output of the SWIM query reformulation engine.

6.2 SWLF Compiler

SWLF Compiler is a module that accomplishes the following processes:

- It transforms the RDF/S virtual schema into Datalog facts that describe the classes, the properties and their subsumption hierarchy (subclasses and subproperties).
- It transforms the RQL user query to a Datalog rule.
- It receives the mappings between RDF/S and XML.

These processes are a preparatory work for the basic procedure of the SWLF compiler, which is to parse the above data and produce the input of the chase/backchase machine.

With the use of JFlex 1.4 [JFl] and Java Cup [CUP] the Datalog facts, rules and the query are parsed to produce an appropriate file for further processing. The full BNF grammar used for parsing the Datalog fragment we employ in our mapping rules and queries is cited in Appendix B.

As discussed thoroughly in Chapter 5 every mapping between RDF/S schema and XML should be translated into constraints. This translation is another responsibility of the SWLF compiler and it is accomplished during the parsing.

6.3 MARS System: A chase/backchase machine

The crucial SWIM component of chase and backchase machine that was discussed previously, is implemented with the help of MARS system [DT03a] [DT03b] [DT02] [DPT01]. MARS is a system implemented in the University of Pennsylvania by Alin Deutsch [Deu02] and Lucian Popa [Pop00].

The MARS system (a system for **M**ixed **A**nd **R**edundant **S**torage) was originally designed for query reformulation and optimization over (object) relational databases. The functionality of MARS relies on checking for query containment and equivalence, and minimizing queries.

MARS uses a compilation of queries, views and constraints from XML into the relational framework. This compilation was proven complete for a large and expressive fragment of XQuery, and it reduces the original reformulation problem to a problem of minimization of relational queries under relational integrity constraints. This problem has been solved by Chase & Backchase (C&B) algorithm which is complete for a large fragment of relational queries and constraints. That means that MARS finds all the existing minimal reformulations.

While earlier approaches on reformulation considered either materialized views or integrity constraints, MARS with C&B algorithm was the first that allowed proving completeness theorems considering both cases.

One of the reasons of its success, was the ability to treat mappings between defined published and proprietary schemas by using both the Global-As-View and Local-As-View integration approaches (GLAV). The way to do that was the interpretation of such mappings as constraints. This constraint-oriented approach is the main technique employed to implement the GLAV integration approach as discussed in Section 5.2.1.

6.4 XPath/XQuery Generator

The module of XPath and XQuery generator is based on the following algorithm: The query (or the union of the queries) produced by the second call of the chase/backchase machine is expressed in the first-order logic predicates used for the representation of XML (GReX: Section 4.2). By examining these predicates we manage to construct a tree as depicted in Figure 5.15. All the information used for this construction is excavated from this query, without any further knowledge.

- Predicates that represent the nodes of the tree are the T, A and E relations.
- The connection between nodes is given either from D (descendant) or C and E (child) relations.
- The content of the nodes is represented by A or Txt relations.

- The root of the query tree is given by XMLroot predicate.

Using the information above, the query tree is constructed and can be represented graphically as shown in Figure 5.15 of the Example 5.12. Once this tree is constructed we traverse it and produce the XPath and/or XQuery expressions.

The process of translation into XPath/XQuery expressions, which is the last step in query reformulation, is done by a Java program.

6.5 Related Work

6.5.1 Styx

Styx [Fun02] is a system for integrating and querying heterogeneous XML sources. The global schema chosen for the integration is an ontology (represented as extended ER schema), defined independently of the local sources schemas by domain or application specialists. This ontology defines concepts and binary relationships between them, which are mapped to the local schemas with the use of XPath expressions. However, Styx uses only a small fragment of XPath expressions: only attribute and child axes. This system adopts the Local-As-View approach instead of SWIM's GLAV one, and additionally, it does not take into account XML constraints.

6.5.2 PEPSINT

PEPSINT [CXH04] is based on a hybrid P2P architecture and it distinguishes between two different types of peers. The super peers that contain an RDF global ontology and the peers that contain local schemas and local data sources. The integration of the sources consists of the following steps: Firstly, an XML to RDF wrapper is used in order to transform the sources' XML schema to a local RDF schema. These local RDF schemas are represented as RDF graphs. The role of the RDF global schema, stored in super peers, is twofold: (i) to provide a uniform view for user queries and (ii) to act as a mediator for the query translation. Additionally,

there is a mapping table that stores mappings between the local schemas and the global ontology.

Despite using paths for the mapping process, PEPSINT maps only single classes or properties and it is not able to map views. This is the reason that PEPSINT adopts the GAV approach instead of SWIM's GLAV. Furthermore, the mappings specified in PEPSINT are RDF→RDF mappings instead of the XML→RDF used in SWIM. However, recall that RDF→RDF mappings can be also specified in SWIM through the RVL view specification language.

An extension of PEPSINT system is presented in [XC04]. In this system relational databases and XML sources are translated into local RDF ontologies and local data are also stored in an RDF repository (RSSDB [ACK+01]). Mappings between the local ontologies are established with the help of a mapping language called RDFMS. RDFMS is a meta-ontology language that represents different semantic relationships between concepts by using types of mappings that declare synonyms, homonyms, hyponyms, unions or intersections of concepts. Each peer receives an RQL user query, which is translated into RQL queries appropriate for the linked peers, by using the ICS-FORTH SWIM.

6.5.3 Clio

Clio [MHH+01] [MHH00] [Vel05] is a research prototype of a schema mapping creation tool. Their focus is on discovering queries that map values from the data sources to values at the global (target) schema. Both source and target schemas are considered to be either relational or XML. In this way, Clio system is able to transform the data, which are structured according to the source schema, to data structured according to the target one. This translation is done in advance, in contrast to SWIM's approach where it is performed on-the-fly for every query.

Clio produces a set of mappings between the source schema and the target one, given a set of high-level correspondences defined by the user. These mappings are expressed in SQL or XQuery and the system explores a large search space in order to find the mappings that minimize the information loss. Additionally, it exploits

constraint information (i.e., keys and foreign keys) from the data sources.

Clio provides the user with an efficient mechanism for establishing GLAV mappings between schemas. Its mapping management tool relieves users from the arduous and time-consuming work of generating and managing mappings and it could be beneficial for SWIM to incorporate such techniques in order to achieve the translation of the sources to RDF data.

6.5.4 Piazza

Piazza [TIM+03] [TH04] [HIMT03] is a peer data management system that enables sharing heterogeneous data. Rather than requiring global agreement, the Piazza system, provides query answering capabilities over an arbitrary network of local schemas and pairwise mappings between them. There are two kinds of mappings: one that relates two peer schemas (peer description) and one that relates a stored schema with a peer schema (storage description). Mappings are specified in an XQuery-like based mapping language, where the path expressions use only child axis. Piazza considers mappings between peers containing XML documents with different structures or XML serializations of RDF descriptions.

Queries are posed over a specific peer schema and are reformulated as follows: Query rewriting algorithms are used for translating the queries in terms of other peers and query unfolding (with the help of storage descriptions) is used, as a last step, for translating the result in terms of stored schema. Queries are expressed as XQueries independently of what the peer schema may be.

Piazza system mappings differ from SWIM's ones in the fact that they use GAV, instead of GLAV, formalisms. However, since they require some LAV features, they support situations where the local schema is a projection or a selection of the global one. In this way they combine the important properties of both LAV and GAV. Additionally, Piazza considers RDF data sources through their XML serialization, without being able to capture the semantics of these sources. For example, its mapping language is not able to capture subsumption relationship between concepts.

6.5.5 RDF Based Architecture for Semantic Integration of Heterogeneous Information Sources

[VH01] presents an architecture for integrating information sources (relational or object databases, HTML pages, XML documents or RDF based sources). All these heterogeneous sources can be integrated, in this approach, by using XML as an intermediate layer between them and the ontology. The XML serialization of sources' data actually creates XML sources, which are integrated by a Concept Model. This Concept Model is expressed in RDF which can be extended with other ontology level features. Mappings between XML and RDF are expressed as rules, specified in LMX. Furthermore, the architecture provides the user with the ability to query the Concept Model and to define F- Logic rules on it.

This architecture constitutes a simplified approach of a semantic integration system, since the problems of data serialization, semantics of mappings or query answering are not answered. Additionally, the rules inference engine launched as an added-value for the architecture demands the retrieval and transformation of the data in the sources in order to infer new facts.

6.5.6 YACOB

Another work that is closely related to SWIM is presented in [SGS03]. YACOB is a mediator that uses domain knowledge in the form of concepts for formulating and processing queries. As a concept model they use an ontology represented in RDF/S, and extent it by introducing some special kinds of RDF/S classes (they introduce categories which in contrast to concepts have no associated extension). Each mapping defines the correspondence of an RDF schema concept or property to a path that determines the XML element of the source's data, and is expressed as XSLT rule. The RDF/S schema can be queried with the help of a query language called CQuery (a derivative of XQuery).

YACOB is a complete enough mediator for integrating XML sources since it provides users with query functionality, as long as, cashing of results. It considers three

types of mappings (adopting the GAV approach), according to what will be mapped (concept, property or category). These mappings should be complete (all the leaf concepts should be mapped to correspondent XML paths).

6.5.7 Integration of XML sources with OWL

This approach [LF04] integrates XML sources through global ontologies expressed in OWL. OWL is used for a double purpose. Firstly, it is used as the global schema language and secondly as semantic mapping language. This is because OWL offers a way to define union or intersection between concepts and properties, as well as, to define equivalence between them and inverse properties. Using these features OWL can be used to express mappings between XML and OWL. The OWL ontology, used for global schema, can be queried with an XQuery-like language, called SWQL.

This integration system follows the Global-As-View instead or our GLAV approach. It's mapping rules map XML Schema constructs (instead of path expressions) to OWL concepts (instead of views over the RDF global schema).

6.5.8 WEESA - WEb Engineering for Semantic web Applications

WEESA [RJG04] is able to produce an RDF representation of a given XML document. In other words, this approach provides a mechanism for republishing an XML document as an RDF description. Given an XML schema, WEESA defines mappings between this schema and an ontology (represented in OWL). These mappings, expressed in OWL, are used to automatically generate the RDF descriptions from an XML document that conforms to that XML schema.

During mapping definitions XPath expressions are used in order to specify the content for the RDF triples. XPath expressions use the child and the attribute axis and in case that more than one XPath expressions are needed (for computing a mapping for a property), specific Java methods are used. This renders mapping specification difficult since many mapping rules should be specifically programmed in

low-level code.

Additionally, SWIM, as will be discussed in Section 7.1, may act as a system for republishing the XML sources, but furthermore it is able to query them.

6.5.9 Interpreting XML via an RDF Schema

The effort of [Kle02] is toward turning XML documents into knowledge structures by interpreting them as RDF data. Starting from a given XML document and an ontology that describes the document (represented in RDF) a set of RDF triples is produced. A domain expert specifies which labels of the XML document are interesting and what is their correspondences in the ontology (a concept or a property). Additionally, they introduce two new RDF properties in order to represent the nesting of the XML document (i.e., property rdfx:value is used to connect a literal value to an element that is matched to a class).

This system tries to associate meaning with an XML document. The basic procedure is a transformation of an XML document to RDF statements and it depends on specific (for the XML document) ontology.

Chapter 7

Conclusion and Future Work

The vision of the Semantic Web is to have data on the web defined and linked in a way that they can be used by machines not just for display purposes, but for reusing and integrating them across various applications. This vision will certainly lead to changes on how we use the Web today. However, there are many obstacles in realizing it, which appear in the inter-operation of SW sources with non-SW ones. It is true that the vast majority of sources are legacy systems that range from relational database systems to virtual or native XML sources. Thus, integrating such data sources to build advanced SW applications is compulsory.

In this thesis, we have presented the SWIM system, a system for integrating XML and relational [Ser05] data sources on the Semantic Web. We addressed a set of problems concerning the integration of XML sources under the peculiarities of the SW and we contributed to the resolution of them.

SWIM relies on a robust framework for SW data integration applications. This is done through viewing the XML and the RDF/S, which is the SW language used for the ontology description, as first-order logic predicates. This relational representation, of both languages, is the first step towards bridging the gap between the XML and RDF/S data models. In this relational framework the semantics of both XML and RDF/S can be captured with the help of constraints, which are expressed as disjunctive embedded dependencies (DEDs).

Additionally, our approach takes benefit from background theory on relational

semantic query optimization. For a large class of queries, problems like query containment, minimization, composition and rewriting with views are solvable.

SWIM provides a mechanism for reformulating queries expressed in terms of a mediated RDF/S schema. These queries, which can be formulated in a declarative RDF/S query and view language RQL/RVL, can be composed with XML to RDF (or RDB to RDF) mappings in order to conclude to XPath and/or XQuery queries. This reformulation is proved to be sound and complete with the help of the Chase and Backchase algorithm. Furthermore, the relational representation of XML is advantageous for expressing (as constraints) information originating from the XML sources. This information can further improve query reformulation by eliminating redundant predicates and queries.

By taking a look at the proposed framework we can discern that it is a data integration system that adopts the GLAV approach in source (local and global) descriptions. This is a beneficial feature, since the expressiveness of GLAV approach exceeds the one of Global-As-View and Local-As-View. Extending the expressiveness of the framework results in more effective mappings between the different worlds.

7.1 Future Work

The functionality of SWIM presented in this thesis can be extended in several ways.

Firstly, the SWIM's functionality can be extended in order to support more expressive SW ontology constraints. This concerns additional features like those offered by description-logic based SW languages, like OWL. For example, it will be beneficial if we could express cardinality constraints or inverse properties.

Additionally, SWIM can incorporate another fragment of XML constraints, called *structural* constraints [BFSW01] [AFL02], in order to further improve query reformulation. These constraints concern the structure of an XML document. For example, a mapping of the form "Name(x):-{/.../Artist/Name}(x)." in the presence of a constraint that declares that "Name" can only be an "Artist's" child, can be reduced

to "Name(x):-{//Name}(x).". This simplifies further computations. The challenge is whether we can express such constraints with the use of disjunctive embedded dependencies (DEDs).

Another important improvement is the ability to return to the user fully typed RDF descriptions according to the RDF/S mediated schema employed by the users to formulate their queries. The results that are currently returned to the user are RDF descriptions that include the resources that consist the answer to his query. However, returning descriptions that have full typing information, will be an added-value functionality and it will exploit the power of the RDF/S language.

The issue of caching previous RQL query results should also be investigated as an improvement for our middleware. In other words, let's assume that SWIM is able to store instances for the virtual mediated RDF/S schema. When the user poses a new query against the RDF/S schema we should be able to decide if this query can be answered by the stored results. By using these results we avoid the reformulation and the execution (at the local sources) of the user's query. This can be achieved by deciding the containment of this query to the materialized one. Our framework is rich enough to support this functionality, by using RDF to RDF mappings.

The next question that arises is whether the proposed middleware is capable of minimize the mappings themselves. The challenge is if mappings can be considered as queries and as such to be processed by middleware. In this way, we could eliminate redundant mappings. For example, if a user specifies the mappings of an "Artist" and a "Painter" to be equal, and we know that "Painter" is a subclass of "Artist", then "Artist's" mapping is redundant. Furthermore, we can infer mappings from existing ones. A query that asks information about "Artists" (and we do not have a mapping for them) can use the mapping that specifies those "Artists" who exhibit in a "Museum" (the mapping for the property "exhibits", since the domain of this property is the class "Artist"). In this way, the user receives part of the information: the "Artists" who have exhibited in "Museums", instead of all the "Artists", since such information is not available.

Finally, there will be an additional advantage if we are able to generate entire

translation programs for SW Data Warehouses as discussed in [MHH+01] [MHH00]. We can achieve this by excavating all the information stored in XML sources, which can be accessed through the RDF/S schema, and storing it as RDF/S descriptions. The republication of entire XML sources will facilitate the construction of SW warehouses with pre-stored RDF data. Such warehouses promote data exchange between groups using different technologies.

Bibliography

[ABS00] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, 2000.

[ACK+01] S. Alexaki, V. Christophides, G. Karvounarakis, D. Plexousakis, and K. Tolle. The ICS-FORTH RDFSuite: Managing Voluminous RDF Description Bases. In *Proceedings of the 2nd International Workshop on the Semantic Web (SemWeb 2001)*, 2001.

[AFL02] M. Arenas, W. Fan, and L. Libkin. What's Hard about XML Schema Constraints. In *Proceedings of the 13th International Conference, DEXA, Aix-en-Provence, France*, 2002.

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison - Wesley, 1995.

[AQM+96] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel Query Language for Semistructured Data. *International Journal of Digital Libraries*, 1996.

[BB04] L. Bertossi and L. Bravo. *Inconsistency Tolerance in Knowledgebases, Databases and Software Specifications*, chapter Consistent Query Answers in Virtual Data Integration Systems. Springer, 2004.

[BFSW01] P. Buneman, W. Fan, J. Simeon, and S. Weinstein. Constraints for Semistructured Data and XML. *SIGMOD Record*, 30(1), 2001.

[BGL+99] C. Baru, A. Gupta, B. Ludaescher, R. Marciano, Y. Papakonstantinou, and P. Velikhov. XML-Based Information Mediation with MIX. In *Exhibition Program of ACM SIGMOD Int. Conference on Management of Data, June*, 1999.

[BLK+04] M. Boyd, C. Lazanitis, S. Kittivoraviktul, P. Mc Brien, and N. Rizopoulos. An Overview of The AutoMed Repository. Technical report, Department of Computing, Imperial College, London SW7 2AZ, 2004.

[CCGL02] A. Cali, D. Calvanese, G. De Giacomo, and M. Lenzerini. Data Integration under Integrity Constraints. In *Proceedings of the International Conference on Advanced Information Systems Engineering (CAiSE)*, 2002.

[CDF+03] D. Chamberlin, D. Draper, M. Fernandez, M. Kay, J. Robie, M. Rys, J. Simeon, J. Tivy, and P. Wadler. *XQuery from the Experts: A Guide to the W3C XML Query Language*. Addison-Wesley, 2003.

[CdGL02] D. Calvanese, G. de Giacomo, and M. Lenzerini. A Framework for Ontology Integration. In *The Emerging Semantic Web, Selected Papers from the First Semantic Web Working Symposium*, pages 201–214, 2002.

[CGMH+94] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The TSIMMIS Project: Integration of heterogeneous information sources. In *Proceedings of IPSJ Conference, Tokyo, Japan, October*, 1994.

[CKK+03] V. Christophides, G. Karvounarakis, I. Koffina, G. Kokkinidis, A. Magkanaraki, D. Plexousakis, G. Serfiotis, and V. Tannen. The ICS-FORTH SWIM: A Powerful Semantic Web Integration Middleware. In *Proceedings of the First International Workshop on Semantic Web and Databases, Humboldt-Universitat, Berlin, Germany, September 7-8*, 2003.

[CKT01] C. Convey, O. Karpenko, and N. Tatbul. Data Integration Services, 2001.

[CLR03] A. Cali, D. Lembo, and R. Rosati. On the decidability and complexity of query answering over inconsistent and incomplete databases. In *Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 2003.

[CUP] CUP Parser Generator for Java. http://www.cs.princeton.edu/∼appel/modern/java/CUP/.

[CVV01] S. Cluet, P. Veltri, and D. Vodislav. View in large scale XML repository. In *Proceedings of the International Conference on Very Large Databases (VLDB), Rome, Italy, September*, 2001.

[CXH04] I. F. Cruz, H. Xiao, and F. Hsu. Peer-to-Peer Semantic Integration of XML and RDF Data Sources. In *The 3rd International Workshop on Agents and Peer-to-Peer Computing (AP2PC 2004)*, July 2004.

[Dee] The Deep Web. http://www.brightplanet.com/deepcontent/index.asp.

[Deu02] A. Deutsch. *XML Query Reformulation Over Mixed and Redundant Storage*. PhD thesis, University of Pennsylvania, 2002.

[DG97] O. Duschka and M. Genesereth. Query Planning in Infomaster. In *Proceedings of the International Conference on Management of Data, SIGMOD, San Jose, CA, USA*, pages 109–111, 1997.

[DGL00] O. Duschka, M. Genesereth, and A. Levy. Recursive Query Plans for Data Integration. *Journal of Logic Programming*, 43(1):49–73, 2000.

[DLW01] D. Draper, A. Levy, and D. Weld. The Nibble System. In *Proceedings of the International Conference on Database Engineering*, 2001.

[DPT99] A. Deutsch, L. Popa, and V. Tannen. Physical Data Independence, Constraints and Optimization with Universal Plans. In *Proceedings of the*

International Conference on Very Large Databases(VLDB), September, 1999.

[DPT01] A. Deutsch, L. Popa, and V. Tannen. Chase & Backchase: A Method for Query Optimization with Materialized Views and Integrity Constraints. Technical report, University of Pennsylvania, 2001.

[DT01] A. Deutsch and V. Tannen. Containment and Integrity Constraints for XPath Fragments. In *Proceedings of the KRDB*, 2001.

[DT02] A. Deutsch and V. Tannen. Querying XML with Mixed and Redundant Storage. Technical report, University of Pennsylvania, 2002.

[DT03a] A. Deutsch and V. Tannen. MARS: A System for Publishing XML from Mixed and Redundant Storage. In *Proceedings of the 29th VLDB Conference, Berlin, Germany*, 2003.

[DT03b] A. Deutsch and V. Tannen. Reformulation of XML Queries and Constraints. In *Proceedings of the International Conference on Database Theory (ICDT)*, 2003.

[DTD] DTD Tutorial. http://www.w3schools.com/dtd/default.asp.

[Dus97] O. Duschka. *Query Planning and Optimization in Information Integration.* PhD thesis, Stanford University, 1997.

[FKPT04] R. Fagin, P. Kolaitis, L. Popa, and W. Tan. Composing Schema Mappings: Second-Order Dependencies to the Rescue. In *Proceedings of the 23th ACM Symposium on Principles of Database Systems, Paris*, 2004.

[FLM99] M. Friedman, A. Levy, and T. Millstein. Navigational Plans for Data Integration. In *Proceedings of the sixteenth national conference on artificial intelligence and eleventh innovation applications of AI conference on Artificial intelligence and innovative applications of artificial intelligence*, pages 67–73, 1999.

[Fun02] I. Fundulaki. *Integration et Interrogations de Resources XML pour Communautes Web*. PhD thesis, CNAM Paris, 2002.

[GMUW01] H. Garcia-Molina, J. D. Ullman, and J. D. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2001.

[HIMT03] A. Halevy, Z. Ives, P. Mork, and I. Tatarinov. Piazza: Data management infrastructure for semantic web applications. In *Proceedings of the 12th Intl. World Wide Web Conference, Budapest, Hungary*, 2003.

[HST99] T. Harder, G. Sauter, and J. Thomas. The intrinsic problems of structural heterogeneity and an approach to their solution. *The VLDB Journal*, 8:25–43, 1999.

[JFl] JFlex - The Fast Scanner Generator for Java. http://jflex.de/.

[JTMP04] E. Jasper, N. Tong, P. McBrien, and A. Poulovassilis. Generating and Optimising Views from Both as View Data Integration Rules. In *Proceedings of 6th Baltic Conference on Database and Information Systems (DBIS'04), Riga, June*, 2004.

[KAC+02] G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl. RQL: A Declarative Query Language for RDF. In *Proceedings of the 11th International World Wide Web Conference (WWW)*, Honolulu, Hawaii, 2002.

[KFvHH01] M. Klein, D. Fensel, F. van Harmelen, and I. Horrocks. The Relation between Ontologies and XML Schemas, 2001.

[Kle02] M. Klein. Interpreting XML via an RDF Schema. In *Proceedings of the 13th International Workshop on Database and Expert Systems Applications (DEXA 2002)*, 2002.

[KLSS95] T. Kirk, A. Levy, Y. Sagiv, and D. Srivastava. The Information Manifold. In *Proceedings of the AAAI Spring Symposium on Information*

Gathering in Distributed Heterogeneous Environments, Stanford, CA, March, 1995.

[KS91] W. Kim and J. Seo. Classifying Schematic and Data Heterogeneity in Multidatabase Systems. *IEEE Computer*, 24(12):12–18, 1991.

[KSC+04] I. Koffina, G. Serfiotis, V. Christophides, V. Tannen, and A. Deutsch. Integrating XML Data Sources using RDF/S Schemas: The ICS-FORTH Semantic Web Integration Middleware (SWIM). Invited Talk: Dagstuhl Seminar (2004): Semantic Interoperability and Integration, September 2004. http://www.dagstuhl.de/04391/.

[Len02] M. Lenzerini. Data Integration: A Theoretical Perspective. In *Proceedings of the ACM PODS, Madison, Wisconsin, USA, 3-6 June*, 2002.

[Lev99] A. Y. Levy. Logic-Based Techniques in Data Integration. In Jack Minker, editor, *Workshop on Logic-Based Artificial Intelligence, Washington, DC, June 14-16*, College Park, Maryland, 1999. Computer Science Department, University of Maryland.

[Lev01] A. Levy. Answering Queries Using Views: A Survey. *The International Journal on Very Large Data Bases*, 2001.

[LF04] P. Lehti and P. Fankhauser. XML Data Integration with OWL: Experiences and Challenges. In *Proceedings of the 2004 International Symposium on Applications and the Internet (SAINT'04)*, 2004.

[LLR02] D. Lembo, M. Lenzerini, and R. Rosati. Review on models and systems for information integration. Technical report, Universita di Roma "La Sapienza", 2002.

[LRO96] A. Levy, A. Rajaraman, and J. J. Ordille. Querying Heterogeneous Information Sources Using Source Descriptions. In *Proceedings of International Conference on Very Large Databases (VLDB), Bombay, India, September*, pages 251–262, 1996.

[LSK95] A. Levy, D. Srivastava, and T. Kirk. Data Model and Query Evaluation in Global Information Systems. *Journal of Intelligent Information Systems*, 1995.

[Mag03] A. Magkanaraki. A View Definition Language for RDF/S. Master's thesis, University of Crete, 2003. Available in Greek only.

[MC02] H. Meisel and E. Compatangelo. EER-ConcepTool: a "reasonable" environment for schema and ontology sharing. In *Proceedings of the 14th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'02), Washington, DC*, 2002.

[MFK01] I. Manolescu, D. Florescu, and D. Kossmann. Answering XML Queries over Heterogeneous Data Sources. In *Proceedings of the 27th VLDB Conference, Roma, Italy*, 2001.

[MH03] J. Madhavan and A. Halevy. Composing Mappings Among Data Sources. In *Proceedings of the 29th International Conference on Very Large Databases(VLDB), Berlin, Germany*, 2003.

[MHH00] R. J. Miller, L. Haas, and M. Hernandez. Clio: Schema Mapping as Query Discovery. In *Proceedings of the 26th VLDB Conference, Cairo, Egypt*, 2000.

[MHH+01] R. J. Miller, M. Hernandez, L. Haas, L. Yan, C. T. Howard Ho, R. Fagin, and L. Popa. The Clio Project: Managing Heterogeneity. *SIGMOD Record*, 30(1), 2001.

[Mit99] P. Mitra. Algorithms for Answering Queries Efficiently Using Views. Technical report, Infolab, Stanford University, September, 1999.

[MTCP03] A. Magkanaraki, V. Tannen, V. Christophides, and D. Plexousakis. Viewing the Semantic Web Through RVL Lenses. In *Proceedings of the Second International Semantic Web Conference (ISWC'03), Sanibel Island, Florida, USA, 20-23 October*, 2003.

[Myl04] J. Mylopoulos. Data Semantics Revisited: Databases and the Semantic Web. In *Proceedings of the DASFAA'04, March 17-19, Jeju Island, Korea*, 2004.

[OWL] OWL Web Ontology Language. http://www.w3.org/TR/2004/REC-owl-ref-20040210/.

[PL00] R. Pottinger and A. Levy. A Scalable Algorithm for Answering Queries Using Views. In *Proceedings of the International Conference on Very Large Databases (VLDB), Cairo, Egypt, September*, 2000.

[PMW96] Y. Papakonstantinou, H. Garcia Molina, and J. Widom. Object Exchange across Heterogeneous Information Sources. In *Proceedings of the International Conference on Data Engineering (ICDE), March*, 1996.

[Pop00] L. Popa. *Object/Relational Query Optimization with Chase and Backchase*. PhD thesis, University of Pennsylvania, 2000.

[rdfa] RDF Primer. http://www.w3.org/TR/rdf-primer/.

[rdfb] RDF/XML Syntax Specification. http://www.w3.org/TR/rdf-syntax-grammar/.

[RJG04] G. Reif, M. Jazayeri, and H. Gall. Towards Semantic Web Engineering: WEESA - Mapping XML Schema to Ontologies. In *Proceedings of the WWW2004 Workshop on Application Design, Development and Implementation Issues in the Semantic Web, New York*, 2004.

[Ser05] G. Serfiotis. Optimizing and Reformulating RQL Queries on the Semantic Web: The ICS-FORTH SWIM. Master's thesis, University of Crete, 2005.

[SGS03] K. Sattler, I. Geist, and E. Schallehn. Concept-based Querying in Mediator Systems. Technical report, Fakultat fur Informatik, Universitat Magdeburg, 2003.

[SK92] A. Sheth and V. Kashyap. So Far (Schematically) yet So Near (Semantically). In *Proceedings of the IFIP WG 2.6 Database Semantics Conference on Interoperable Database Systems (DS-5)*, 1992.

[SY80] Y. Sagiv and M. Yannakakis. Equivalences among Relational Expressions with the Union and Difference Operators. *Journal of the ACM*, 27:633–655, 1980.

[TH04] I. Tatarinov and A. Halevy. Efficient Query Reformulation in Peer Data Management Systems. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, 2004.

[TIM+03] I. Tatarinov, Z. Ives, Y. Madhavan, A. Halevy, D. Suciu, N. Dalvi, X. (Luna) Dong, Y. Kadiyska, G. Miklau, and P. Mork. The Piazza Peer Data Management Project. *SIGMOD Record*, 32(3), 2003.

[Ull00] J. Ullman. Information Integration Using Logical Views. *Theoretical Computer Science*, 239(2):189–210, 2000.

[Vel05] Yannis Velegrakis. *Managing Schema Mappings in Highly Heterogeneous Environments*. PhD thesis, University of Torondo, 2005.

[VH01] R. Vdovjak and G. Houben. RDF Based Architecture for Semantic Integration of Heterogeneous Information Sources. In *Proceedings of the International Workshop on Information Integration on the Web - Rio de Janeiro, RJ, Brasil*, 2001.

[XC04] H. Xiao and I. Cruz. RDF-based Metadata Management in Peer-to-Peer Systems. In *Proceedings of the 2nd IST Workshop on Metadata Management in Grid and P2P Systems (MMGPS), London*, 2004.

[XMLa] Extensible Markup Language (XML) 1.0 (Third Edition). http://www.w3.org/TR/2004/REC-xml-20040204/.

[XMLb] XML Schema. http://www.w3.org/TR/xmlschema-0/.

[XPaa] XML Path Language (XPath) 1.0. http://www.w3.org/TR/xpath.

[XPab] XML Path Language (XPath) 2.0. http://www.w3.org/TR/xpath20/.

[XQu] XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/.

Appendix A

RQL Translation Patterns

This appendix shows how we translate the core RQL patterns into conjunctions of the first order predicates that are used for the representation of RDF. Firstly, we consider the RQL class patterns.

Class Pattern	Translation
$C ^$C	CLASS(c)
$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)
$C{X}	C_SUB(d, c), C_EXT(d, x)
^$C{X}	C_EXT(c, x)
$C{$D} ^$C{$D}	C_SUB(d, c)

Next we consider the RQL property patterns.

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

Appendix B

BNF grammar for Datalog Rules

[1]	PROGRAM	::=	FACTLIST RULELIST QUERY
			;
[2]	FACTLIST	::=	FACT FACTLIST
			|
			;
[3]	FACT	::=	CLASS_FACT
			| PROP_FACT
			| PSUB_FACT
			| CSUB_FACT
			;
[4]	CLASS_FACT	::=	*"CLASS("* CONSTANT *")"* *"."*
			;
[5]	PROP_FACT	::=	*"PROP("* TRI_CONSTANT *")"* *"."*
			;
[6]	PSUB_FACT	::=	*"PSUB("* DBL_CONSTANT *")"* *"."*
			;
[7]	CSUB_FACT	::=	*"CSUB("* DBL_CONSTANT *")"* *"."*
			;
[8]	TRI_CONSTANT	::=	CONSTANT *","* CONSTANT *","*

```
                                    CONSTANT
                                    ;
[9]  DBL_CONSTANT          ::=  CONSTANT "," CONSTANT
                                    ;
[10] RULELIST              ::=  RULE RULELIST
                                    |
                                    ;
[11] RULE                  ::=  HEAD ":-" XQBODY "."
                                    ;
[12] XQBODY                ::=  "{" XPATH "}" "("
                                    XQVAR_LIST ")" "," XQBODY
                                    | "{" XPATH "}" "(" XQVAR_LIST
                                    ")" "," COMP_LIST "|"
                                    FUNCTION_LIST
                                    | "{" XPATH "}" "(" XQVAR_LIST
                                    ")" "," COMP_LIST
                                    | "{" XPATH "}" "(" XQVAR_LIST
                                    ")" "|" FUNCTION_LIST
                                    | "{" XPATH "}" "(" XQVAR_LIST
                                    ")"
                                    ;
[13] XQVAR_LIST            ::=  VARIABLE "," VARIABLE
                                    | VARIABLE
                                    ;
[14] COMP_LIST             ::=  COMPAR "," COMP_LIST
                                    | COMPAR
                                    ;
[15] FUNCTION_LIST         ::=  FUNC "," FUNCTION_LIST
                                    | FUNC
                                    ;
```

[16] HEAD ::= HEAD_PREDICATE *","* HEAD
| HEAD_PREDICATE
;

[17] HEAD_PREDICATE ::= CONSTANT *"("* VARIABLE *")"*
| CONSTANT *"("* VARIABLE *","*
VARIABLE *")"*
;

[18] DECIMALLITERAL ::= NUMBER
;

[19] STRINGLITERAL ::= STRING_VALUE
;

[20] NCNAME ::= CONSTANT
| VARIABLE
;

[21] QNAME ::= PREFIX *":"* LOCALPART
| LOCALPART
;

[22] PREFIX ::= NCNAME
;

[23] LOCALPART ::= NCNAME
;

[24] XPATH ::= EXPR
;

[25] EXPR ::= EXPRSINGLE EXPR_ADDED
;

[26] EXPR_ADDED ::= *","* EXPRSINGLE EXPR_ADDED
|
;

[27] EXPRSINGLE ::= OREXPR
;

[28] OREXPR ::= ANDEXPR
;

[29] ANDEXPR ::= INSTANCEOFEXPR
ANDEXPR_ADDED
;

[30] ANDEXPR_ADDED ::= *"and"* INSTANCEOFEXPR
ANDEXPR_ADDED
|
;

[31] INSTANCEOFEXPR ::= TREATEXPR
;

[32] TREATEXPR ::= CASTABLEEXPR
;

[33] CASTABLEEXPR ::= CASTEXPR
;

[34] CASTEXPR ::= COMPARISONEXPR
;

[35] COMPARISONEXPR ::= RANGEEXPR COMP_ADDED
RANGEEXPR
| RANGEEXPR
;

[36] COMP_ADDED ::= VALUECOMP
| GENERALCOMP
| NODECOMP
;

[37] RANGEEXPR ::= ADDITIVEEXPR
;

[38] ADDITIVEEXPR ::= MULTIPLICATIVEEXPR
;

[39] MULTIPLICATIVEEXPR ::= UNARYEXPR

;

[40] UNARYEXPR ::= UNIONEXPR

;

[41] UNIONEXPR ::= INTERSECTEXCEPTEXPR
UNIONEXPR_ADDED

;

[42] UNIONEXPR_ADDED ::= *"|"* INTERSECTEXCEPTEXPR
UNIOEXPR_ADDED
|

;

[43] INTERSECTEXCEPTEXPR ::= VALUEEXPR

;

[44] VALUEEXPR ::= PATHEXPR

;

[45] PATHEXPR ::= *"/"*
| *"//"* RELATIVEPATHEXPR
| *"//"* RELATIVEPATHEXPR
| RELATIVEPATHEXPR

;

[46] FUNCTIONPATHEXPR ::= *"/"* RELATIVEPATHEXPR
| *"//"* RELATIVEPATHEXPR

;

[47] RELATIVEPATHEXPR ::= STEPEXPR
RELATIVEPATHEXPR_ADDED

;

[48] RELATIVEPATHEXPR_ADDED ::= *"/"* STEPEXPR
RELATIVEPATHEXPR_ADDED
| *"//"* STEPEXPR
RELATIVEPATHEXPR_ADDED
|

;

[49] STEPEXPR ::= AXISSTEP
| FILTERSTEP
;

[50] AXISSTEP ::= FORWARDSTEP PREDICATES
| REVERSESTEP PREDICATES
;

[51] FILTERSTEP ::= PRIMARYEXPR PREDICATES
;

[52] CONTEXTITEMEXPR ::= *“.”*
;

[53] PRIMARYEXPR ::= LITERAL
| CONTEXTITEMEXPR
;

[54] PREDICATES ::= *“[”* EXPR *“]”* PREDICATES
|
;

[55] GENERALCOMP ::= *“=”*
| *“!=”*
;

[56] VALUECOMP ::= *“eq”*
| *“neq”*
;

[57] NODECOMP ::= *“is”*
| *“isnot”*
;

[58] FORWARDSTEP ::= FORWARDAXIS NODETEST
| ABBREVFORWARDSTEP
;

[59] REVERSESTEP ::= REVERSEAXIS NODETEST

| ABBREVREVERSESTEP
;

[60] ABBREVFORWARDSTEP ::= *“@”* NODETEST
| NODETEST
;

[61] ABBREVREVERSESTEP ::= *“..”*
;

[62] FORWARDAXIS ::= *“child” “::”*
| *“”descendant” “::”*
| *“attribute” “::”*
| *“self” “::”*
| *“descendant-or-self” “::”*
;

[63] REVERSEAXIS ::= *“parent” “::”*
| *“ancestor” “::”*
| *“ancestor-or-self” “::”*
;

[64] NODETEST ::= KINDTEST
| NAMETEST
;

[65] NAMETEST ::= QNAME
| WILDCARD
;

[66] WILDCARD ::= *“*”*
;

[67] LITERAL ::= NUMERICALLITERAL
| STRINGLITERAL
;

[68] NUMERICALLITERAL ::= DECIMALLITERAL
;

[69] KINDTEST ::= TEXTTEST
| ANYKINDTEST
;

[70] TEXT_TEST ::= *"text()"*
;

[71] ANYKINDTEST ::= *"node()"*
;

[72] COMPAR ::= EQUALITY
;

[73] EQUALITY ::= VARIABLE *"="* VALUE
| VARIABLE *"="* VARIABLE
;

[74] FUNC ::= CONCAT_F
;

[75] CONCAT_F ::= *"myConcat"* *"("* VARIABLE *","*
MIXED_CONCAT *","*
MIXED_CONCAT *")"*
;

[76] MIXED_CONCAT ::= VARIABLE
| VALUE
| FUNCTIONPATHEXPR
;

[77] VALUE ::= STRING_VALUE
| NUMBER
;

[78] QUERY ::= Q_HEAD *":-"* Q_BODY *"."*
;

[79] Q_HEAD ::= *"QUERY"* *"("* VARIABLE_LIST *")"*
;

[80] VARIABLE_LIST ::= VARIABLE

| VARIABLE *","* VARIABLE_LIST

;

[81] Q_BODY ::= QITEM

| Q_ITEM *","* Q_BODY

;

[82] QITEM ::= COMPAR

| *"P_EXT("* TRP_PEXT *")"*

| *"P_SUB("* DBL_PSUB *")"*

| *"PROP("* TRP_PROP *")"*

| *"C_EXT("* DBL_CEXT *")"*

| *"C_SUB("* DBL_CSUB *")"*

;

[83] VARCONST ::= VARIABLE

| CONSTANT

;

[84] DBL_CSUB ::= VARCONST *","* VARCONST

;

[85] DBL_PSUB ::= VARCONST *","* VARCONST

;

[86] DBL_CEXT ::= VARCONST *","* VARIABLE

;

[87] TRP_PEXT ::= VARIABLE *","* VARCONST *","*
VARIABLE

;

[88] TRP_PROP ::= VARIABLE *","* CONSTANT *","*
VARIABLE

;