UNIVERSITÉ DE BOURGOGNE

**THÈSE**

pour obtenir le grade de

**Docteur de l'Université de Bourgogne**

Spécialité : **Informatique**

par

**Raji GHAWI**

15 / 03 / 2010

Directeur de thèse: **Nadine Cullot**

# Ontology-based Cooperation of Information Systems

## Contributions to Database-to-Ontology Mapping and XML-to-Ontology Mapping

# *Abstract*

This thesis treats the area of ontology-based cooperation of information systems. We propose a global architecture called OWSCIS that is based on ontologies and web-services for the cooperation of distributed heterogeneous information systems. In this thesis, we focus on the problem of connecting the local information sources to the local ontologies within OWSCIS architecture. This problem is articulated by three main axes: 1) the creation of the local ontology from the local information sources, 2) the mapping of local information sources to an existing local ontology, and 3) the translation of queries over the local ontologies into queries over local information sources.

**Keywords**: Information Systems, Ontologies, Databases, Mapping, XML, OWL, SPARQL, SQL, XQuery, Query Translation.

---

# *Résumé*

Cette thèse traite le domaine de coopération des systèmes d'informations basée sur les ontologies. Nous proposons une architecture globale, appelée OWSCIS, qui se base sur les ontologies et les services-web pour la coopération des systèmes d'informations distribués et hétérogènes. Dans cette thèse, nous focalisons sur la problématique de connexion des sources d'informations locales vers des ontologies locales dans le cadre de l'architecture OWSCIS. Cette problématique est articulée en trois axes principaux: 1) la création de l'ontologie locale des sources d'informations locales, 2) la mise en correspondance des sources d'informations locales avec l'ontologie locale, et 3) la traduction des requêtes sur l'ontologie locale vers des requêtes sur les sources d'informations locales.

**Mots Clés**: Systèmes d'Informations, Ontologies, Bases de Données, Mise en Correspondance, XML, OWL, SPARQL, SQL, XQuery, Traduction de Requêtes.

# *Acknowledgements*

I am grateful to many people for help, both direct and indirect, in writing this thesis.

My biggest thanks go to my advisor Prof. Nadine Cullot. She was always with me, and gave me a lot of her time, care and patience. This thesis would never have become reality without her continuous help and support.

I would also like to thank the boss of our research team Prof. Kokou Yetongnon. His comments and suggestions have encouraged, supported and enlightened me.

I am grateful to the jury members: M. Stefano Spaccapietra, M. Guy Pierra and M. Gilles Dubois, for the care with which they reviewed this thesis.

I want to express my gratitude to the professors and doctors in the Databases research team of the LE2I laboratory. Especially, I would like to thank Prof. Christophe Nicole for his gracious support. He was always encouraging me to hard work on writing research papers. Many thanks go to Richard Chbier, Christophe Cruz, Lylia Abrouk and David Gros-Amblard for their support and cooperation.

I want to express my gratitude to my friends, family and colleagues, whose support and good have kept me going through the writing of this thesis.

I would particularly like to thank my partners in OWSCIS project, Thibault Poulain and Guillermo Gomez, for their valuable collaboration.

I would also like to thank my colleagues in LE2I laboratory, especially: Fekade Getahun, Bechara Al Bouna, Joe Tekli, Elie Abi Lahoud, Elie Raad, Monica Ferreira, Sylvain Rakotomalala, Hyunja Lee, Duan Yucong, Gilbert Tekli, Damien Leprovost and Aries Muslim for their continuous help and support.

*to my darling wife, Abir*

*to my sweet children, Khaled and Arwa*

*and to my dear parents*

# Contents

# List of Figures

# List of Tables

# Chapter 0

# Introduction

Today, huge amounts of information are becoming available over the web and over corporate and governmental networks. The exponential growth of the Internet as well as current advances in telecommunications technologies led to an unparalleled increase of the number of online information sources. However, the access to all information available remains limited as long as information is stored separately without easy means to combine them from different sources. The information society requires a complete and easy access to all available information. This exacerbates the need for suitable methods to combine data from different sources.

The cooperation of information systems is the ability to share, combine and/or exchange information between multiple agents (individuals, companies and governments) and/or from multiple information sources, and the ability to access the integrated information by their final receivers transparently.

The major problems that hinder the cooperation of information systems are: the autonomy, the distribution, the heterogeneity and the instability of information sources. In particular, we are interested by the heterogeneity problem that can be identified over several levels: the system, the syntactic, the structural and the semantic heterogeneity.

The cooperation of information systems has been extensively studied and several approaches have been proposed to bridge the gap among heterogeneous information systems, such as: database-translation, standardization, federation, mediation and web-services.

These approaches provide pertinent solutions to the heterogeneity problem at syntactic and structural levels. However, in order to achieve semantic interoperability between heterogeneous information systems, the meaning of the information that is interchanged has to be understood across the systems. Semantic conflicts occur whenever two contexts do not use the same interpretation of the information. Therefore, in order to deal with semantic heterogeneity there is a need for more semantic-specialized approaches, such as ontologies.

In this dissertation, we focus on the usage of ontologies as a means for the cooperation of information systems at the semantic level. We propose a global architecture called OWSCIS[1] that is based on ontologies and web-services for the cooperation of distributed heterogeneous information systems.

OWSCIS architecture uses ontologies to represent the semantics of information sources. These information sources can be structured, such as relational databases, and/or semi-structured, such as XML documents. However, information sources are normally distributed over multiple separated sites, where each site can contain several information sources. Therefore, at each site, a local ontology is used to describe the semantics of the local information sources. In addition, OWSCIS uses a global ontology as a pivot global model for all participating local ontologies.

In this dissertation, we focus on the problem of connecting the local information sources to the local ontologies. Such connection includes three main axes:

1. the creation of the local ontology from the local information sources,
2. the mapping of local information sources to an existing local ontology, and
3. the translation of queries over the local ontologies into queries over local information sources.

[1]**O**ntology and **W**eb **S**ervice based **C**ooperation of **I**nformation **S**ources

The problem of connecting the local information sources to the local ontologies also relates to the type of information sources in consideration (databases and XML data sources). Considering the three axes mentioned above with each of these two types of information sources, we obtain six problems to solve. In this dissertation, we try to solve most of these six problems as shown in the following table:

| | ontology creation | mapping to existing ontology | query translation |
|---|---|---|---|
| single database | yes (chapter 7) | yes (chapter 7) | yes (chapter 8) |
| single XML data source | yes (chapter 10) | no | yes (chapter 11) |

This thesis is organized in four parts distributed over 11 chapters:

The first part introduces the topic of information integration using ontologies. It comprises two chapters:

- Chapter 1 gives a general background on the area of cooperation of information systems.
- Chapter 2 gives a background on the area of ontology-based cooperation of information systems, and a survey on the existing approaches in this area.

The second part is dedicated to the presentation of OWSCIS architecture. It consists of two chapters:

- Chapter 3 contains an overview on OWSCIS architecture and its main components.
- Chapter 4 summarizes the contributions held in the dissertation within OWSCIS architecture.

The third part treats the area of database-to-ontology mapping. It includes four chapters:

- Chapter 5 gives a background on the topic of database-to-ontology mapping, including the main issues addressed in this topic, and the existing works in the literature.
- Chapter 6 presents our proposed database-to-ontology mapping specifications that are "Associations with SQL statements" and "DOML language".
- Chapter 7 presents our proposed tool DB2OWL whose purposes are: the ontology creation from a single database, and the mapping of a single database to an existing ontology. This tool supports both of our mapping specifications proposed in Chapter 6.
- Chapter 8 is devoted to the problem of SPARQL-to-SQL query translation. We propose two translation methods that are based on our mapping specifications.

The fourth part treats the area of XML-to-ontology mapping. It includes three chapters:

- Chapter 9 gives a background on the topic of XML-to-ontology mapping and the existing works in the literature.
- Chapter 10 presents our contributions to the topic of XML-to-ontology mapping. We propose a mapping specification called XOML, and a tool called X2OWL that aims at the ontology creation from a single XML data source.
- Chapter 11 introduces our proposed method for SPARQL-to-XQuery query translation that is based on our XOML mapping specification.

Finally, the last chapter concludes the dissertation and presents the perspectives and future works.

This dissertation is closed by a set of appendices that present further backgrounds, examples and algorithms. Readers can find some interseting details in these appendices.

# Chapter 1

# Background

**Contents**

## Abstract

In this chapter, we give a general background on the topic of information integration (also known as: *interoperability of information systems*).

Firstly, we discuss the need for the interoperability of information systems and the different dimensions of this problem, namely: the autonomy, distribution, heterogeneity and instability of information systems.

In particular, we address the heterogeneity dimension of the interoperability problem, and we present the different levels of this dimension: system, syntactic, structural and semantic heterogeneities.

Then, we review some of the approaches proposed in the literature to tackle the heterogeneity problem, including: database-translation, standardization, federation, mediation and web-services.

Finally, we discuss the need for more semantic-specialized approaches, such as ontologies, in order to deal with semantic heterogeneity.

## 1.1 Interoperability

### 1.1.1 What is the Interoperability ?

In the last few years, huge amounts of information are becoming available over the web and over corporate and governmental networks. The exponential growth of the Internet as well as current advances in telecommunications technologies led to an unparalleled increase of the number of online information sources. However, the access to all information available remains limited as long as information is stored separately without easy means to combine them from different sources. The necessity of the information society to a complete and easy access to all information available exacerbates the need for suitable methods to combine data from different sources.

Even at a single organization level, decision makers often need information from multiple, disparate sources, including databases, file systems, digital libraries, and electronic mail systems, but are unable to get and fuse the required information in a timely fashion due to the difficulties of accessing the different systems, and due to the fact that the information obtained can be inconsistent and contradictory [69].

In general terms, *information integration* is the ability to share, combine and/or exchange information between multiple agents (individuals, companies and governments) and/or from multiple information sources, and the ability to access the integrated information by their final receivers transparently. This problem is also known as the *interoperability problem.* The Institute of Electrical and Electronic Engineers (IEEE) provides the generally accepted definition of interoperability as, [97]:

**Definition 1.1. Interoperability** is the ability of two or more systems or components to exchange information and to use the information that has been exchanged.

The interoperability problem has recently gained increasing attention for several reasons, including:

1. Increasing number of independently created and managed information sources.
2. Increasing specialization of work, and its subsequent need to reuse and analyze data.
3. Large variety of data representation paradigms.
4. Rising acquisition costs of complex non-traditional data (e.g. multimedia data).

From enterprise point of view, information integration unifies different views of the information landscape. While developers use a technical-oriented model to express their view, managers try to capture the concepts behind each application. An integrated view of the information resources of an enterprise has various benefits, [2]:

**Data discovery** – it is easer to locate the storage place of relevant business information, to identify how it links to other information stored at a different location, and to discover where (intended or unintended) redundancies are.

**Data integration** – accessing and manipulating disparate data sources can be supported and automated through a unified view. The impact of changes can be accessed more easily.

**Data quality improvement** – in a unified view, data consistency can be verified and managed more easily because the enforcement of consistency rules can be centralized. Furthermore, data cleansing, which means removing data inconsistencies between redundant information sources, is simpler.

### 1.1.2 Why the interoperability is difficult to achieve?

Interoperability remains a difficult problem due to several reasons and creates new challenges for many areas of computer science. The first difficulty is to find suitable information source that might contain data needed for a given task. This problem is addressed in the areas of information retrieval and information filtering [14]. Despite this, the major difficulties to achieve efficient interoperability are: autonomy, distribution, heterogeneity and instability of information sources [37].

#### Autonomy

The users and the applications can access to the data through a federated system or by their own local system. When the information system bases on multiple sites (called components or component systems), the autonomy that integrated components may retain, becomes a critical issue. The autonomy can be classified in three types ([39][129]):

**Design autonomy** means that a component is independent from others in its design, regarding issues such as its universe of discourse, data model, naming concepts and so on. Design autonomy also entails the autonomy to change the design at any point in time, which is particularly difficult to handle within infrastructures of interoperating components.

**Communication autonomy** is given when a component can decide independently with which other systems it communicates. Within federations of interoperating components communication autonomy means that each component can leave and enter the federation at any time.

**Execution autonomy** denotes the component independence in execution and scheduling of incoming requests. Execution autonomy is almost impossible to retain if a global transaction management is involved.

### Distribution

Another difficulty to integrate data sources is their physical distribution. Since most computers are nowadays connected to some type of network, especially the Internet, it is natural to think of combining application and data sources that are physically located on different hosts, but that can communicate through the network.

The distribution is problematic for interoperability when searched information is distributed over multiple information sources. That is, a query could not be answered by the data available from a single information source, but the answer is broken into fragments that are distributed among physically distinct sources.

### Heterogeneity

The most difficult obstacle for information integration is the heterogeneity of information sources to be integrated. Different sources might have different data models, require different access methods, and even be hosted by different hardware platforms and operating systems.

Heterogeneity naturally occurs due to the fact that an autonomous development of systems always yields different solutions. Many reasons lie behind this fact, such as 1) different understanding and modelling of the same real-world concepts, 2) the technical environment, 3) particular requirements on the application, such as high performance for special queries, etc. [39]. In the next section we will discuss the different types of heterogeneity in details.

### Instability

Finally, the fourth complication is instability. New information sources appear every day, others disappear. The format and contents of the sources may change over time. A new attribute may appear, or an existing attribute may be dropped

In an information world characterized by distribution, heterogeneity and instability, computer science researchers strive to provide suitable solutions to achieve efficient interoperability.

Research on the integration of heterogeneous sources aims at recognizing and combining relevant knowledge so as to provide a richer understanding of a particular domain. The integration is particularly valuable if it enables the communication between different sources allowing them to maintain their autonomy. Current research efforts concentrate on defining models, methods, architectures, and tools that make it easier to integrate information from distributed, heterogeneous information sources in an instable environment [58].

In the next section, we review in details the heterogeneity dimension of interoperability and its different levels.

## 1.2 Heterogeneity Levels

Bridging heterogeneity is one of the main tasks of information integration. The heterogeneity dimension of the interoperability problem has been well documented. In the literature, there are many classifications of heterogeneity on different levels of detail. They sometimes coincide and sometimes differ. We will follow the classification given by Sheth in [143], where four levels of heterogeneity are identified: the system, syntactic, structural and semantic heterogeneity.

### 1.2.1 System level

This level concerns differences in the technical aspects, like hardware platforms and operating systems (hardware heterogeneity), and communication systems. The system heterogeneity is further classified in [143] into :

**Information System Heterogeneity** – includes the heterogeneity of database management systems, data models, system capabilities such as concurrency control and recovery.

**Platform Heterogeneity** – includes operation system related aspects (heterogeneity of file system, naming, file types, operation, transaction support), and hardware related aspects (heterogeneity of instruction set, data representation/coding).

Researchers and developers have been working on resolving such heterogeneity for many years. During the second half of the 1970s, we saw the ability to deal with hardware, operating systems, and communications heterogeneity.

### 1.2.2 Syntactic level

This level is related to the choices of data models, languages and representations. It involves differences in machine-readable aspects of data representation, including 1) data model heterogeneity, 2) interface heterogeneity, and 3) format heterogeneity.

**Data model heterogeneity** The information may be modelled according to different paradigms, for example as relational database tables or as hierarchical trees in XML. Data model heterogeneity captures the fact that different data models have different semantics for their concepts. For instance, the relational model has no inheritance in contrast to object-oriented models.

**Interface heterogeneity** This type of heterogeneity exists if different components are accessible through different access methods. This includes:

- Language heterogeneity: different query languages (e.g. SQL for relational databases, OQL for object-oriented databases etc.) and language restrictions (e.g. no negation, no disjunctive conditions etc.)
- Query restrictions: only certain queries are allowed, only certain conditions can be expressed, joins can only have up to a certain number of relations, certain attribute values must be specified to form a valid query (binding restrictions), etc.

**Format heterogeneity** This type of heterogeneity involves the problems due to different data types (e.g. short integer vs. integer and/or long), and to different data formats (e.g. 02-04-2004 vs. 02/04/04.)

### 1.2.3 Structural level

The structural level is related to the choices of modelling. It includes *representational heterogeneity* that involves data modelling constructs, and *schematic heterogeneity* that particularly appears in structured databases.

#### Representational heterogeneity

This type of heterogeneity is due to discrepancies in the way related information is modeled/represented in different schemas. We present two common examples of such conflicts:

1. Attribute composition

The same information can be represented either by a single attribute (e.g., as a composite value) or by a set of attributes, possibly organized in a hierarchical manner.

Another classical example of attribute composition conflict is: one source represents person names by a long string that contains both the first and the last name, while another source represents names using last name and first name attributes.

2. Same attribute in different structure

The same attribute may be located in different positions in different schemas.

#### Schematic heterogeneity

This type is related to the encoding of concepts using different elements of a data model [100]. In the relational model, there are three types of such conflicts, [121]:

- Relation vs. attribute name
- Attribute name vs. attribute value
- Relation vs. attribute value

### 1.2.4 Semantic level

The semantic level involves the meaning of schema elements (e.g., classes, attributes and methods). Semantic conflicts occur whenever two contexts do not use the same interpretation of the information. John Sowa defines semantics in [147] as following:

**Definition 1.2.** Semantics determines how the constants and the variables are associated with things in application domain.

Semantics is often used in contrast with *syntax*. On the one hand, syntax is used to refer to the definition of the structure of schema elements, and it is considered to be context independent. On the other hand, semantics is the people's interpretation according to their understanding of the world and therefore, context dependent [87].

Semantic level of heterogeneity concerns the meaning of the data and schema, and thus passes beyond previous levels. Even schemas formulated in the same model can have different semantics.

Let us assume a schema consisting of relations and attributes. These relations and attributes have names and carry an implicit semantic, which is the **concept** they stand for. However, the schema contains in first place only the name, but not the concept. The interpretation of names by different people does not necessarily coincide.

Cui et al. in [50] consider that the semantic heterogeneity deals with three types of concepts:

Semantically equivalent concepts - In this case, a model uses different terms to refer the same concept, e.g. synonymous, or some properties are modelled differently by different systems, for example the concept length may be "meter" in one system and "mile" in one another.

Semantically unrelated concepts - In this case, the same term may be used by different systems to denote completely different concepts.

Semantically related concepts - In this case, different classifications may be performed, for example one system classifies "person" as "male" and "female" and other system as "student" and "professor".

Dealing with different concepts and interpretations causes different semantic conflicts. Goh identifies three main causes for semantic heterogeneity [76]:

1. **Confounding conflicts** – These conflicts occur when information items seem to have the same meaning, but differ in reality, e.g. due to different temporal contexts.
2. **Scaling and units conflicts** – This type of conflicts refer to the adoption of different units of measures or scales in reporting. For example, financial data are routinely reported on different currencies and scale-factors. Also, academic grades may be reported on several different scales having different granularities (e.g., a five point scale with letter grades A, B, C, D and F, or a four-point scale comprising excellent, good, pass and fail).
3. **Naming conflicts** – These conflicts occur when naming schemes of information differ significantly. Frequent phenomena of naming conflicts are homonyms (when the same term is used to refer to different concepts) and synonyms (where different terms are used to refer to the same concept), and diverging understanding of a concept itself, etc.

***Summary***

In this section, we have reviewed the heterogeneity dimension of interoperability problem. We have seen that four levels of heterogeneity (namely, system, syntactic, structural and semantic) are identified in the literature.

In the next section, we will see the different solutions that are propoesed in the literature to tackle the problem of information system heterogeneity.

## 1.3 Interoperability Approaches

Information system interoperation have been extensively studied in the past. Several approaches have been proposed to bridge the gap among heterogeneous information systems.

### 1.3.1 Database-Translation

The database translation approach is a point-to-point solution based on direct data mappings between pairs of information systems. The mappings are used to resolve data discrepancies among the systems [4][26][45][170]. This approach is appropriate when the number of participants in the interoperability environment is small. The number of data translators grows with the square of the number of components in the integrated system.

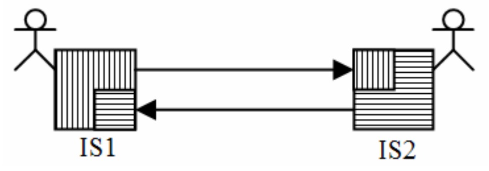


Figure 1.1: Database-Translation approach

In Figure 1.1, a subset of the data from information systems IS1 is translated into a subset of data in the information system IS2. Likewise, a subset of the data of IS2 is mapped into IS1.

### 1.3.2 Standardization

In the standardization approach, the components of the interoperability environment use the same model or standard for data representation and communication. The standard model can be a comprehensive meta-model capable of integrating the requirements of the models of the different components [6][10][93]. The use of a standard meta model reduces the number of translators (this number grows linearly with the number of components) used to resolve semantic differences among components. However, the construction of a comprehensive meta-model is a difficult task; the manipulation of high level languages is complex; and there is no unified database interfaces.

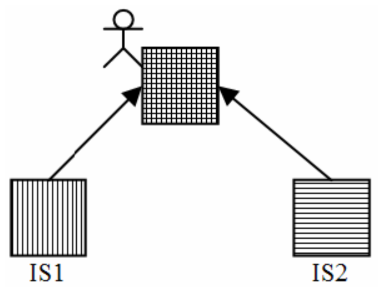


Figure 1.2: Standardization approach

In Figure 1.2, a centralized information system can be built to replace the original information systems (IS1, IS2). The global centralized schema is a combination of the entire data contained in IS1 and IS2.

### 1.3.3 Federation

The federation approach consists of an integrated collection of heterogeneous databases in which federation users access and manipulate data transparently without knowledge of data location

[144]. A federation contains a federated schema that incorporates the data exported by one or more remote information systems (the federation participants/components).

Two types of federations can be distinguished. Tightly coupled federations use a global federated schema constructed by federation administrator to combine the schemas of all participant information systems.

The second type is the loosely coupled federation. It is based on one or more non-global federated schemas created by local users or database administrators for specific purposes. The federated schema incorporates a subset of the schema available in the federation. This approach becomes rapidly complex when the number of translators required becomes large.

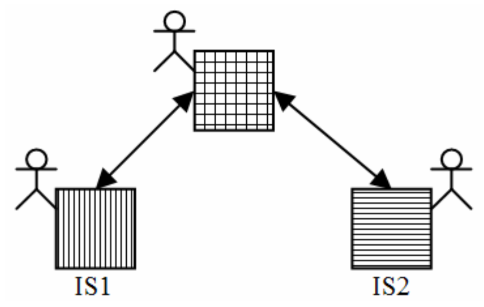


FIGURE 1.3: Federation approach

In Figure 1.3, the existing information systems are completely operational for local users. Only the shared data are integrated in the federated schema. The federated system is only made of information that IS1 and IS2 want to exchange.

Sheth [144] proposed to extend the classical three-level schema architecture of database systems to a five-level schema architecture [144] that supports the distribution, heterogeneity and autonomy of federated database systems. The five-level schema architecture of Sheth includes the following layers (Figure 1.4):

1. Local Schema – This layer includes local schemas of participant information sources (components). Each local schema is expressed using its native data model.

2. Component Schema – A component schema is derived by translating local schema into a data model called the canonical or common data model. The translation is done using transforming processors that generate suitable mappings between objects of local and component schemas. Component schemas are useful because they allow to describe heterogeneous local schemas using a single representation, and because semantics that are missing in a local schema can be incorporated in its component schema. Thus, they facilitate integration tasks in tightly coupled FDBS, and specification of views in loosely coupled FDBS.

3. Export Schema – An export schema represents a subset of a component schema that is available to the federation (the subset that the participant information system wants to share with others). It may include access control information regarding its use by specific federation users. Access control is provided by filtering processors. The export schema help in managing flow of control of data.

4. Federated Schema – A federated schema is an integration of multiple export schemas. It includes information on data distribution that is generated when integrating export schemas.

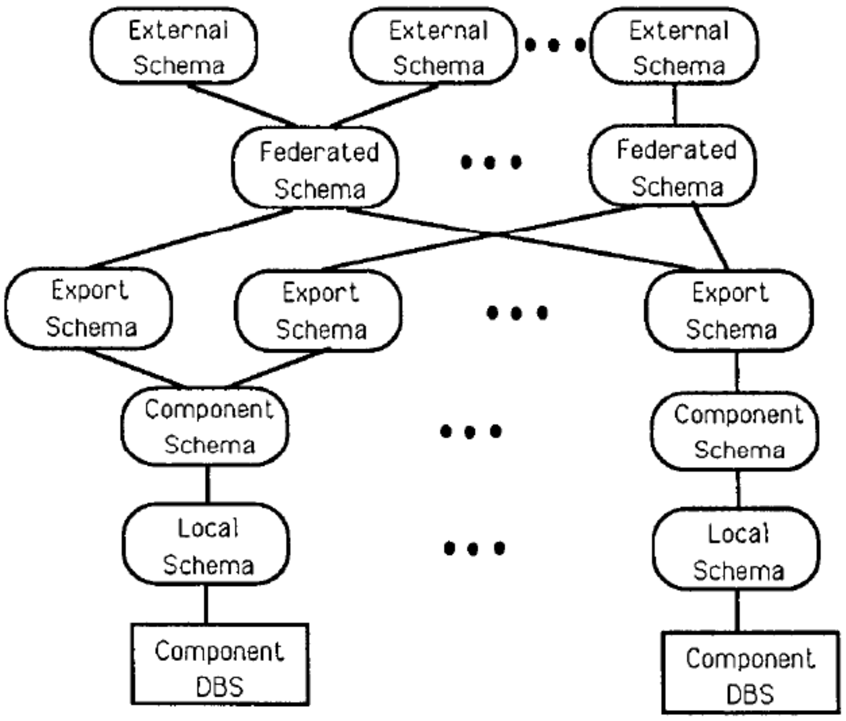


FIGURE 1.4: Five-level schema architecture of an FDBS

Constructing processors are used to transform commands on the federated schema into commands on export schemas.

5. External Schema – An external schema defines a schema for a user and/or application or a class of users/applications. External schemas are useful to customize information in federated schemas, to specify additional integrity constraints, and provide additional access control.

The different layers of this architecture support the different features of the interoperability problem. Firstly, component schemas and transofrming processors support the heterogeneity feature. Then, export schemas and filtering processors support the autonomy feature. Finally, federated schemas and constructing processors support the distribution feature.

### 1.3.4 Language-based Multi-Base

The language based multi-database approach consists of a loosely coupled collection of databases in which a common query language (often SQL-like) is used to access the contents of the participating databases [98][108]. The common language provides constructs that permit queries involving several databases at the same time, as well as operators for users to perform resolution of semantic conflicts. In this approach, in contrast to the distributed and federated systems, no predefined or partial global schema is used. Instead, the users must discover and understand the semantics of other information systems. This approach lacks distribution and location transparency for users.

In Figure 1.5, the various components have to define a global common language (Q) to query their information systems (IS1, IS2). This solution is well adapted for information systems that are based on the same family of data models and do not require complex query translators.

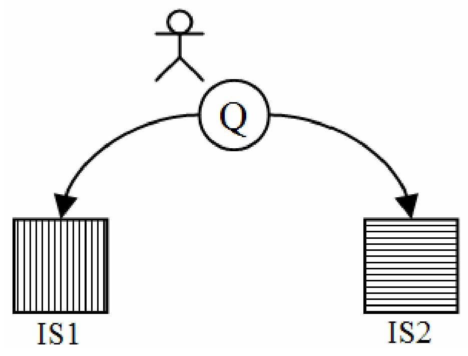


FIGURE 1.5: Language based multi-base approach

### 1.3.5 Mediation

The mediation approach is an extension of the federated approach that is suitable for large and dynamic environments. By dynamic environment we mean an information space with 1) large number of participant information systems (components), 2) frequently updated components, and 3) dynamic addition and removal of components.

The purpose of the mediation approach is to provide an interoperability solution that combines the transparency of tightly-coupled federation and the flexibility of loosely-coupled federation. In this approach, information sources are highly autonomous, and does not only contain structured sources (databases), but also semi-structured sources (e.g., XML files) and unstructured sources (e.g., text files).

The mediation approach is based on two architectural tiers: 1) mediator-tier that is responsible for interfacing with applications and end-users, and 2) wrapper-tier that is responsible for interfacing with information sources.

The *mediator* is used to create and support an integrated view of data from multiple sources. That is, in mediation approach there is no single global schema; but each mediator provides a federated schema. The mediator also provides data discovery support and various query processing services. In particular, it combines, integrates, and abstracts the information provided by the sources. Normally the sources are encapsulated by wrappers.

The *wrapper* is used to map the local databases into a common federation data model. It provides the basic data access functions [69]. In particular, the wrapper translates mediator requests to requests at the information sources.

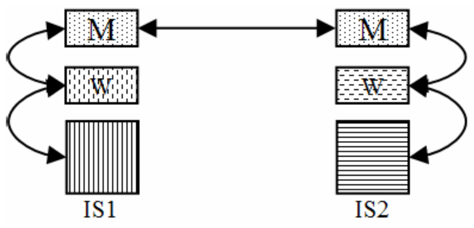


FIGURE 1.6: Mediator-wrapper approach

In Figure 1.6, a translator, which acts as a wrapper, is placed between the conceptual representation of the mediator and the local description of each information source.

Mediator-tier can be hierarchical, i.e., a mediator can send request to other mediators as well. For instance a mediator can cooperate with other mediators to decompose a query into sub queries and generates an execution plan based on the resources of the cooperating sites.

Wrappers are a standard technology to overcome heterogeneity on the syntactic level, since they hide the internal data structure model of a source and transform its contents to a uniform data structure model [159]. At the other hand, mediators are used to overcome heterogeneity on the structural level, since they integrate heterogeneous structures by defining mapping rules between different information structures. The task of structural data integration is, to re-format the data structures to a new homogeneous data structure. This can be done with the help of a formalism that is able to construct one specific information source out of numerous other information sources. This is a classical task of a middleware which can be done with CORBA [128] on a low level or rule-based mediators [166] on a higher level.

The rules in the mediator describe how information of the sources can be mapped to the integrated view. In simple cases, a rule mediator converts the information of the sources into information on the integrated view. The mediator uses the rules to split the query, which is formulated with respect to the integrated view, into several sub-queries for each source and combine the results according to query plan.

A mediator has to solve the same problems which are discussed in the federated database research area, i.e. structural heterogeneity (schematic heterogeneity) [100] and semantic heterogeneity (data heterogeneity) [101][164]. In rule-based mediators, rules are mainly designed in order to reconcile structural heterogeneity, while discovering semantic heterogeneity problems and their reconciliation play a subordinate role.

### 1.3.6 Web-services

The last evolution of interoperability solutions is the Web-services. They are defined as programmable components of loosely coupled and distributed applications accessible on the Web using standard Internet protocols [161]. For the W3C, Web-services are software applications identified by an URI, where interfaces and links are capable to be defined, described and discovered by XML objects. In this definition, Web-services support direct interactions with other software applications using XML based messages.

Web services have been recently proposed as a method to address some of the challenges of web-based integrated systems. A web service can be viewed as a set of layers contained in a stack. The layers are dynamically defined following user needs and are called through a set of Internet protocols. The base set of protocols is composed of: SOAP [162], WSDL [163] and UDDI [157]. They allow the discovery, description and information exchanges between web services.

In order to achieve interoperability of heterogeneous information systems, a set of web services can be built from each information system independently from the other information systems. The web services become a standard interface to access the local information system. These web services can be used by customers and partners via the internet and by local users via an intranet (Figure 1.7). This solution is very flexible and reduces the complexity of the heterogeneity problem [124][146].

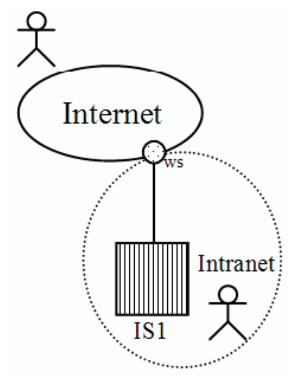


Figure 1.7: Web-Services approach

## 1.4 Semantics and Ontologies

Integrating information from disparate applications is still a time- and money-consuming activity. The expense is not because the information is on different platforms or in different formats, but because of semantic differences between the applications. Discrepancies in the way information sources are specified hinder information sharing between software applications. Conversely, being able to explicitly model the meaning of information guarantees to move information integration towards flexibility and automation.

We have seen in the previous section that the different approaches that have been proposed to achieve interoperability provide pertinent solutions to the heterogeneity problrm at syntactic and structural levels. However, in order to achieve semantic interoperability between heterogeneous information systems, the meaning of the information that is interchanged has to be understood across the systems. Semantic conflicts occur whenever two contexts do not use the same interpretation of the information.

Ontologies provide a promised technology to solve the semantic heterogeneity problem. The term *ontology* was introduced by Gruber [83] as an "explicit specification of a conceptualization". A *conceptualization*, in this definition, refers to an abstract model of how people commonly think about a real thing in the world; and *explicit specification* means that concepts and relationships of an abstract model receive explicit names and definitions.

In general terms, an ontology is an organization of a set of terms related to a body of knowledge. Unlike a dictionary, which takes terms and provides definitions for them, an ontology organizes a knowledge on a basis of concepts. An ontology expresses a concept in a precise manner that can be interpreted and used by computer systems, whereas dictionary definitions are not processable by computer systems. Another difference is that by relying on concepts and specifying them precisely and formally we get definitive meanings that are independent of language or terminology.

An ontology gives the name and the description of the domain specific entities by using predicates that represent relationships between these entities. The ontology provides a vocabulary to represent and communicate domain knowledge along with a set of relationships containing the vocabulary's terms at a conceptual level. Therefore, because of its potential to describe the semantic of information sources and to solve the heterogeneity problems, an ontology might be used for data integration tasks.

Ontologies have several advantages when used for information integration [38]:

1. The vocabulary provided by the ontology serves as a stable conceptual model to the information sources and is independent of the source schemas,
2. The language used by the ontology is expressive enough to address the complexity of queries.
3. Knowledge represented by the ontology is sufficiently comprehensive to support translation of all the relevant information sources into a common frame of reference.

From enterprise point of view, the intent of the ontology is twofold. At the first hand, the ontology provides a common language to allow all the systems within an enterprise to talk to each other and, eventually, for the enterprise to talk to its trading partners. At the other hand, the ontology represents an pivot data model that unifies the structures of all information sources within the enterprise.

Uschold and Grüninger [158] mention interoperability as a key application of ontologies and many ontology based approaches to information integration in order to achieve interoperability have been developed.

Appendix A gives a detailed introduction to ontologies.

## Chapter Summary

In this chapter, we introduced the problem of information integration (or interoperability). This problem have different dimensions: autonomy, distribution, heterogeneity and instability. In particular, we addressed the heterogeneity dimension. Four levels of information system heterogeneity (namely, system, syntactic, structural and semantic) are identified in the literature.

Several approaches are proposed in the literature to tackle the heterogeneity problem, including database-translation, standardization, federation, mediation and web-services. These approaches provide satisfying solutions to syntactic and structural heterogeneities. However, more semantic-specialized approaches are needed to deal with semantic heterogeneity.

Ontologies are one of efficient technologies to solve the semantic heterogeneity problem. In the context of information integration, ontologies can be used to describe the semantics of information sources and to make their content explicit. In the literature, many ontology-based approaches to information integration have been developed.

The next chapter is devoted to review existing ontology-based integration systems.

# Chapter 2

# Related Works

Contents

## Abstract

As mentioned in Chapter 1, there are many research projects that have been proposed to achieve information integration. The goal of such systems is to permit the exploitation of several independent information sources as if they were a single source, with a single global schema. In other words, information integration systems tend to solve the problems of distribution, heterogeneity and autonomy of information sources (see Section 1.1.2).

Among others, we are interested in information integration systems that are based on ontologies. In this chapter, we give an overview on the field of ontology-based information integration. In fact, it is very difficult to exhaustively cover the state of the art of this field for several reasons including: 1) the large number of proposed systems, and 2) the variety of the contexts within which those approaches are exploited. Therefore, we will firstly review the main aspects that characterize ontology-based information integration systems. Then, we review some of well-known systems in the literature.

## 2.1 Issues in Ontology-based Information Integration

There are several criteria that are usually used to compare ontology-based integration systems, such as: the types of information sources involved in the integration, the architecture type, the use of ontologies, the ontology representation language and the relationship between global and local schemas.

### 2.1.1 Information Sources

The information sources involved in a cooperation system are the sources used by the system to retrieve the information to answer users' queries. Buccella et al. in [38] divide the information sources feature into two categories: the state of information sources (SIS) and the type of information sources (TIS).

The state of information sources (SIS) may be static or dynamic. This issue is related to the instability problem of interoperability (see Section 1.1.2). A dynamic integration system should support instable information sources, i.e. sources that connect or disconnect to the integration platform at any time. In this case, the integration system should provide some means to know which sources are available at a given moment.

The main types of information supported by integration systems include: databases, XML documents, files and HTML pages. Most of existing systems support databases but only some of them support semi-structured data (XML or HTML pages).

### 2.1.2 Architecture Type

Several types of architecture are generally identified in cooperation systems, including: federation-based, agent-based, wrapper/mediator -based, and service-oriented architectures.

Federation-based architecture [144] is based on a federated schema that incorporates the data exported by one or more remote information systems (see Section 1.3.3). Federation architecture is old enough and did not coexist with ontology-based cooperation systems.

Agent architectures are designed to allow software processes to communicate knowledge across networks, in high-level communication protocols. They are highly dynamic and open, allowing agents to locate other agents at runtime, discover the capabilities of other agents, and form cooperative alliances. However, if the number of agents is large, the communication among them may become expensive to implement and their interaction become difficult to understand. Few of ontology-based cooperation systems use agent-based architecture.

As mentioned in Section 1.3.5, mediation-based architectures are based on two main components: mediator and wrapper. The mediator is usually used to create an integrated view of data over multiple sources, and the wrapper is used to map local information sources to a common data model. In this type of architecture, all the information needed to achieve the integration is stored in the mediator. However, the mediator can make itself appear complex and difficult to manipulate. Also, performance aspects must be taken into account. Many ontology-based cooperation systems use mediation-based architecture.

Service-oriented architecture (SOA) [59] is the most recent type of architectures. SOA separates functions into distinct services which are accessible over a network in order that users can combine and reuse them in the production of applications. These services communicate with each other by passing data from one service to another, or by coordinating an activity between two or more services. SOA is not yet adopted in ontology-based data integration systems.

### 2.1.3 Ontology Use

The main role of ontologies in information integration systems is to describe the information sources and to make their content explicit. However, the ontologies may be exploited in different ways. Wache et al. [165] distinguish three types of approaches according to the way of exploiting ontologies in information cooperation: single, multiple and hybrid ontology approaches.

#### Single ontology approaches

This is the simplest approach where one global ontology is used and all information sources are linked to it. The relationships are expressed via mappings that identify the correspondence between each information source and the ontology (Figure 2.1).

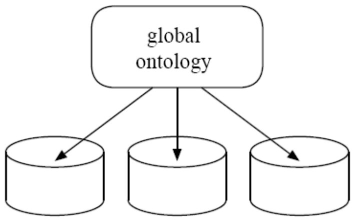


FIGURE 2.1: Single Ontology Approach

The single ontology approach can be applied when all information sources to be integrated provide nearly the same view on a domain. But if one information source has a different view on a domain, e.g. by providing another level of granularity, finding the minimal ontology commitment [84] becomes a difficult task.

Also, single ontology approaches are susceptible for changes in the information sources which can affect the conceptualization of the domain represented in the ontology. That is, changes in one information source can imply changes in the global ontology and in the mappings to the other information sources [150].

These disadvantages led to the development of multiple ontology approaches.

### Multiple ontologies approaches

In this approach, each information source is described by its own ontology and inter-ontology mappings are used to express the relationships between the ontologies (Figure 2.2).

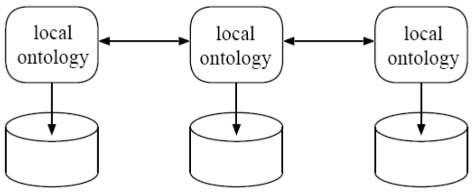


FIGURE 2.2: Multiple Ontology Approach

The advantage of multiple ontology approaches is that no common and minimal ontology commitment [84] about one global ontology is needed. Each source ontology can be developed without respect to other sources or their ontologies. This ontology architecture can simplify the integration task and supports the change, i.e. modifications in one information source or the adding and removing of sources.

On the other hand, the lack of a common vocabulary makes it difficult to compare different source ontologies. To overcome this problem, an additional representation formalism defining the inter-ontology mapping is needed. The inter-ontology mapping identifies semantically corresponding terms of different source ontologies, e.g. which terms are semantically equal or similar. But the mapping has also to consider different views on a domain e.g. different aggregation and granularity of the ontology concepts.

In practice, the inter-ontology mapping is very difficult to define, because of the many semantic heterogeneity problems which may occur.

### Hybrid approaches

Hybrid approaches combine the single and multiple ontology approaches and overcome their drawbacks. Similar to multiple ontology approaches the semantics of each source is described by its own ontology. But in order to make the local ontologies comparable to each other they are built from a global shared vocabulary [76] (Figure 2.3). The shared vocabulary contains basic terms (the primitives) of a domain which are combined in the local ontologies in order to describe more complex semantics. Sometimes the shared vocabulary is also an ontology [159]. In hybrid approaches, two types of mappings are needed: 1) mappings between each information source and its local ontology and 2) mappings between local ontologies and the global ontology.

The advantage of a hybrid approach is that new sources can easily be added without the need of modification. It also supports the acquisition and evolution of ontologies. The use of a shared

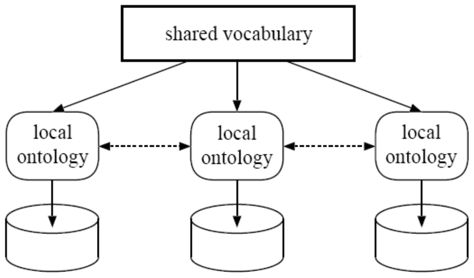


FIGURE 2.3: Hybrid Ontology Approach

vocabulary makes the source ontologies comparable and avoids the disadvantages of multiple ontology approaches. But the drawback of hybrid approaches is that existing ontologies can not easily be reused, but have to be redeveloped from scratch.

### 2.1.4 Representation language

The use of ontologies in the context of information integration is related to the nature of the used ontologies. The nature of ontologies mainly depends on 1) the kind of languages used, and 2) the general structures that can be found.

In the literature, there are two main families of ontology specification languages (Section A.4):

1. Traditional ontology languages (such as KIF, Ontolingua, OKBC, F-Logic, and LOOM).
2. Ontology markup languages (such as RDF(S), XOL, SHOE, DAML+OIL, and OWL).

Ontology specificaion languages are based on different knowledge representation paradigms such as: First order logic, Frame-based systems, and Description Logics. We refer to [47] and [165] for a comparison of ontology languages.

### 2.1.5 Relationship between Global and Local Schemas

Data integration systems can be classified according to the way the schema of the local data sources are related to the global schema. Currently, there are two main initiatives to integrate data and answer queries without materializing a global schema: Global-as-view (GAV) [70] and Local-as-View (LAV) [102].

In order to illustrate these approaches, we use the data integration framework proposed by Lenzerini in [111]. In this framework, a data integration system $\mathcal{I}$ is formalized in terms of a triple $\langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$, where $\mathcal{G}$ is the global schema (or, mediated schema), $\mathcal{S}$ is the source schema, and $\mathcal{M}$ is the mapping between $\mathcal{G}$ and $\mathcal{S}$ constituted by a set of assertions of the forms

$$q_{\mathcal{S}} \rightsquigarrow q_{\mathcal{G}}$$
$$q_{\mathcal{G}} \rightsquigarrow q_{\mathcal{S}}$$

where $q_{\mathcal{S}}$ (resp. $q_{\mathcal{G}}$) is a query expressed over the source schema $\mathcal{S}$ (resp. the global schema $\mathcal{G}$). An assertion $q_{\mathcal{S}} \rightsquigarrow q_{\mathcal{G}}$ specifies that the concept represented by the query $q_{\mathcal{S}}$ over the sources corresponds to the concept in the global schema represented by the query $q_{\mathcal{G}}$.

### Global-As-View (GAV)

The Global-As-View (GAV) approach represents global schema as virtual views over local schemas. That is, in the GAV approach, the mapping $\mathcal{M}$ associates to each element $g$ in $\mathcal{G}$ a query $q_{\mathcal{S}}$ over $\mathcal{S}$. Formally speaking, a GAV mapping is a set of assertions, one for each element $g$ of $\mathcal{G}$, of the form

$$g \rightsquigarrow q_{\mathcal{S}}$$

From the modeling point of view, the GAV approach is based on the idea that the content of each element $g$ of the global schema should be characterized in terms of a view $q_{\mathcal{S}}$ over the sources.

The main advantage of GAV approach is that processing queries expressed over the global schema is easy. In fact, the mapping directly specifies which source queries corresponds to the elements of the global schema, that is, it explicitly tells the system how to use the sources to retrieve data. GAV approach is effective whenever the data integration system is based on a set of sources that is stable. In most GAV systems, query answering is based on a simple unfolding strategy.

However, extending the system with a new source is a problem. In fact, the new source may indeed have an impact on the definition of various elements of the global schema, whose associated views need to be redefined. Thus, the GAV approach is not appropriate in a dynamic environment where it is practically impossible for a site to know all the other sites.

### Local-As-View (LAV)

The Local-As-View (LAV) approach defines local source schemas as views over global mediated schema. That is, in the LAV approach, the mapping $\mathcal{M}$ is a set of assertions, one for each element $s$ of $\mathcal{S}$, of the form

$$s \rightsquigarrow q_{\mathcal{G}}$$

From the modeling point of view, the LAV approach is based on the idea that the content of each source $s$ should be characterized in terms of a view $q_{\mathcal{G}}$ over the global schema. This idea is effective whenever the data integration system is based on a global schema that is stable and well-established in the organization (an enterprise model, or an ontology).

The main advantage of LAV approach is that it favors the extensibility of the system: adding a new source requires only the definition of mappings between the new source and the global schema. This simply means enriching the mapping with a new assertion, without other changes.

However, processing queries in the LAV approach is a difficult task. In fact, the only knowledge we have about the data in the global schema is through the views representing the sources, and such views provide only partial information about the data. Thus, it is not immediate to infer how to use the sources in order to answer queries expressed over the global schema.

Beside GAV and LAV, alternative approaches that use more general types of mappings have been also discussed in the literature, including Global Local-As-View (GLAV) [68], BYU (Brigham Young University)-Global-Local-As-View (BGLAV) [169] and Both-As-View (BAV) [31]. These approaches attempt to combine the LAV and GAV approaches in order to make the best of them. That is, it would be easy to reformulate requests, as in the case of GAV approach, as well as to modify local schemas, as is the case of LAV approach.

In the next section, we will present several ontology-based integration systems which are well-known in the literature. We then conclude this chapter by discussing the issues presented in this section with respect to the systems presented in the next section.

## 2.2 Existing Ontology-based Approaches

In this section, we give an overview on most known ontology-based information integration systems, including: BUSTER, COIN, DOME, InfoSleuth, KRAFT, MOMIS, and OBSERVER.

### 2.2.1 BUSTER

BUSTER system (Bremen University Semantic Translator for Enhanced Retrieval) [159] [151] [152] has been developed at the Center for Computing Technologies at the university of Bremen, Germany.

The entire integration process in BUSTER includes two phases:

#### Data acquisition phase

The aim of this phase is to gather information about the data sources to be integrated. This information is stored in a source-specific data base called the Comprehensive Source Descriptor (CSD). A CSD has to be created for each data source that participates in a network of integrated data sources. Each CSD consists of: 1) metadata that describe technical and administrative details of the data source, 2) structural and syntactic schema and annotations, and 3) a source ontology, (a detailed and computer-readable description of the concepts stored in the data source).

In order for a data source to be able to exchange data with the other data sources in the network, Integration Knowledge (IK) must also be acquired. An IK describes how the information can be transformed from one source to another.

#### Query phase

In this phase, a user or an application formulates a query to an integrated view of sources. Several specialized components in the query phase use the acquired information (CSD, IK) to select the desired data from several information sources and to transform it to the structure and the context of the query.

In the query phase, several components of different levels interact:

At the *syntactic level*, wrappers are used to establish a communication channel to the data source(s), that is independent of specific file formats and system implementations.

At the *structural level*, mediators are responsible for the reconciliation of the structural heterogeneity problems. The mediator uses information obtained from the wrappers and combines, integrates and abstracts them. The mediator is configured by a set of rules that describe in a declarative style, how the data from several sources can be integrated and transformed to the data structure of original source. The rules are acquired in the acquisition phase with the help of the rule generator.

At the *Semantic level*, two different tools specialized for solving the semantic heterogeneity problems are used: functional context transformation and context transformation by re-classification.

### 2.2.2 COIN

The COIN (Context Interchange) Project [145][78][77][34][66] was initiated in 1991 at the Massachusetts Institute of Technology (MIT), USA. Its goal is to achieve semantics interoperability among heterogeneous information sources.

The COIN system provides access to both traditional data sources such as databases and semi-structured information sources such as Web sites. The system has a mediator-based architecture. The main elements of this architecture are wrappers, context axioms, elevation axioms, a domain model, context mediators, an optimizer and an executioner.

COIN introduces a new definition for describing things in the world. It states that the truth of a statement can only be understood with reference to a given context. Here, the notion of context is useful to model statements in conflicting heterogeneous databases. Using this definition, COIN creates a framework that constitutes a formal and logical specification of the COIN system components. This framework comprises three components:

- The *domain model* is a collection of source's primitive types, such as strings or integers, and "rich" types, called semantic types, which defines the application domain corresponding to the data sources to be integrated. Primitive objects and semantics objects are instances of primitive types and semantics types respectively. Semantics objects may have properties called modifiers that serve as annotations to make the data's semantics of different contexts explicit. Also, semantics objects may have different values in different contexts.
- The *elevation axioms* corresponding to each source identify the correspondences between attributes in the source and semantic types in the domain model. In addition, they codify the integrity constraints of the sources, which are useful for producing optimal queries.
- The *context axioms* corresponding to named contexts associated with different sources or receivers, define alternative interpretations of the semantic objects in different contexts. Every source or receiver is associated with exactly one context, but several sources may share the same context.

COIN has a different representation of an ontology but it is based on an hybrid ontology approach. In COIN, a change in one source will generate changes in some components to represent the new mappings. In COIN, elevation and context axioms have to be defined to access the new information. The COIN framework is specified as a deductive and object-oriented data model of the family of F-logics [99].

### 2.2.3 DOME

The DOME (Domain Ontology Management Environment) project [50][49] has been conducted by BTexact Technologies, UK. It is developing techniques for ontology-based information contents description and a suite of tools for domain ontology management. Within this project, a methodology, a set of tools and an architecture are developed to enable enterprise-wide information management for data re-use and knowledge sharing. The system retrieves information from multiple resources to answer user queries and presents the results in a consistent way that is meaningful to the user.

The DOME architecture consists of a number of interacting components: an ontology server which is responsible for managing the definitions of terms, a mapping server which manages the relationships between ontologies, an engineering client with tools for developing and administrating a DOME system, a user client to support querying the knowledge shop, and a query engine for decomposing queries and fusing the results to sub-queries.

### 2.2.4 InfoSleuth

InfoSleuth [11][167][55][67] is an agent-based system for the integration of heterogeneous sources. It is developed by Microelectronics and Computer Technology Corporation (MCC), Austin, Texas, USA. The purpose of the InfoSleuth project is to retrieve and process information in a network of heterogeneous information sources.

In InfoSleuth system, ontologies are used to capture database schemas, conceptual models and aspects of its agent architecture. Here, there are two main tasks to accomplish, 1) describing the information sources, and 2) specifying the agent infrastructure, i.e. the context in which agents operate, its relevant information and relationships, etc.

InfoSleuth system have an agent-based architecture to provide interoperation among autonomous systems. The different sources are integrated in a dynamic way and this is made possible by using a network of co-operating agents that form the InfoSleuth architecture. There are five kinds of agents in InfoSleuth [11]:

**User Agent** – This agent provides an interface that enables the user to communicate with the system independently of location. It obtains information about the ontologies known to the system and it uses them to prompt its user in selecting an ontology that will be used to formulate queries.

**Resource Agent** – This agent allows the InfoSleuth architecture to access the information sources and executes the requests concerning a specific resource. It translates queries expressed in a common query language into a language understood by the resources. This translation comprises both the mapping of the shared ontology into database schema, and the mapping of the query language into the native language.

**Ontology Agent** – This agent is a specialized Resource Agent whose main task is to answer questions about ontologies. It maintains a knowledge base of the different ontologies used for specifying requests. Using this knowledge base, the Ontology Agent can answer queries about the ontologies available, such as the source of an ontology and search the ontologies for concepts.

**Broker Agent** – This agent aims at finding the set of relevant resources that can solve a user query. It collectively maintain the information that the agents advertise about themselves. All InfoSleuth agents advertise their capabilities to the broker agent that semantically matches agents looking for a particular service with agents providing that particular service (*information brokering technique*).

**Task Execution Agent** – This agent use information provided by a broker agent to route requests to the appropriate Resource Agents. It decomposes user queries into sub-queries and reassembles the answers, thus coordinating the executions of high-level information gathering sub-tasks.

### 2.2.5 KRAFT

KRAFT (Knowledge Reuse and Fusion / Transformation) [81][136][137] is a multi-site research project conducted at the universities of Aberdeen, Cardiff and Liverpool in collaboration with BT (British Telecommunications) in the UK.

The overall aim of this project is to enable the sharing and reuse of constraints embedded in heterogeneous databases and knowledge systems. The concept of "Knowledge fusion" is used to denote the combination of knowledge from multiple, distributed, heterogeneous sources in a dynamic way.

The architecture uses constraints as a common knowledge interchange format, expressed against a common ontology. Knowledge held in local sources can be transformed into the common constraint language, and fused with knowledge from other sources. The fused knowledge is then used to solve some problem or deliver some information to a user. Problem-solving in KRAFT typically exploits pre-existing constraint solvers. KRAFT uses an open and flexible agent architecture in which knowledge sources, knowledge fusing entities, and users are all represented by independent software agents, communicating using a messaging protocol.

The KRAFT architecture has the following components:

**User Agent** – is the interface between users and services provided by KRAFT domain.

**Resource** – is the knowledge source to integrate. It provides services to the KRAFT domain. Examples of KRAFT resources are databases, knowledge bases and constraint solvers.

**Wrapper** – is the interface between the domain and the user agent or the resources. Wrappers provide communication services, both at high and at low level. At high level they support the mechanisms linking the resources to mediators and facilitators. At low level they provide a translation service between the internal data formats of users agent and resources and the internal data format supported by the KRAFT domain. They co-operate with the ontology agent to perform translations.

**Mediator** – is the component that retrieves information on a domain. In achieving this purpose it uses domain knowledge to transform data in order to increase its information content. Examples of data transformation might be integration of data from heterogeneous sources, consistency checking and refinement of knowledge and data, etc. The mediator agent performs operations on queries to implement a certain task and can process queries by decomposing, combining them and transforming their content.

**Ontology Agent** – is the component that translates knowledge expressed against a source ontology into the knowledge expressed against a target ontology. If a mediator or a wrapper requires an ontology translation it passes the expression and references to both source and target ontologies to the ontology agent who will translate and return the expression.

**Facilitator** – is the KRAFT component performing the internal routing services for messages within the KRAFT domain. Its main functions are to maintain records of the location and of the capabilities of the resources, and to accept and route messages from other KRAFT resources. When an agent needs a service, it asks a facilitator to recommend another agent that provides the service.

### 2.2.6 MOMIS

MOMIS (Mediator envirOnment for Multiple Information Sources) [16][17] has been conceived as a joint collaboration between University of Milano and Modena in the framework of the INTERDATA Italian research project, aiming at providing methods and tools for data management in Internet-based information systems.

The goal of MOMIS is to give the user a global virtual view of the information coming from heterogeneous information sources. MOMIS creates a global mediation schema (ontology) for the structured and semi-structured heterogeneous data sources, in order to provide the user with a uniform query interface.

The architecture of MOMIS consists of the following components:

- A common *object-oriented data model*, defined according to the ODL$_{I^3}$ language, to describe source schemas for integration purposes. ODL$_{I^3}$ is an object-oriented data-description language with an underlying description logic language OLCD [17], which enables inferences to be made (e.g. subsumption) about the classes expressed in that language. In addition, ODL$_{I^3}$ introduces new constructors to support the semantic integration process
- One or more *wrappers*, to translate schema descriptions into the common ODL$_{I^3}$ representation. In addition, the wrapper performs a translation of the query from the OQL$_{I^3}$ language to a local request to be executed by a single source.
- A *mediator*, which is a software module that combines, integrates, and refines ODL$_{I^3}$ schemata received from the wrappers. In addition, the mediator generates the OQL$_{I^3}$ queries for the wrappers, starting from the query request formulated with respect to the global schema. The mediator consists of a query manager (QM) and a global schema builder (GSB). The QM component breaks up global OQL$_{I^3}$ queries into sub-queries for the different data sources. The mediator module is obtained by coupling a semantic approach, based on description logics component, i.e., ODB-Tools Engine [18] and an extraction and analysis component, i.e., Schema Analyzer and Classifier, together with a minimal ODL$_{I^3}$ interface.
- A *query-processing component*, based on two pre-existing tools, namely ARTEMIS and ODB-Tools, to provide an architecture for integration and query-optimization. The ARTEMIS tool environment performs the classification (affinity and synthesis) of classes for the synthesis of the global classes. The ODB-tools engine performs schema validation and inferences for the generation of the common thesaurus.

### 2.2.7 OBSERVER

OBSERVER (Ontology Based System Enhanced with Relationships for Vocabulary hEterogeneity Resolution) [120][119][80] is a project conducted at the University of Zaragoza, Spain.

The aim of the OBSERVER project is to retrieve and process information stored in heterogeneous knowledge sources (called repositories). OBSERVER was created assuming dynamic information sources. Any type of information source such as HTML pages, databases or files, is supported. OBSERVER uses the concept of data repository, which might be seen as a set of entity types and attributes. Each repository has a specific data organization and may or may not have a data manager. The different data sources of a repository might be distributed.

The heterogeneity in this project concerns paradigms and ontological assumptions. To overcome the differences in the formats and in the languages, OBSERVER relates repositories to domain ontologies; these are pre-existing ontologies defining a set of terms in a specific domain. In OBSERVER, ontologies are described using a system based on Description Logics (DL) such as CLASSIC [28] or LOOM [114][115].

The OBSERVER architecture has several nodes, each one includes query processing capabilities and data repositories accessible to the rest of the nodes through ontologies that describe them. Each node comprises two main elements:

- **Query processor** – This element has as input a user query expressed using terms from a chosen user ontology. The query processor accesses the data repositories to answer the query. If the user is not satisfied with the answer, the query processor translates (partially or totally) the query into another user-selected ontology using predefined inter-ontology relationships. The query processor generates a list of translation plans, where each plan has an associated loss of information.
- **Ontology server** – Every node has only one Ontology Server which is a module that provides information about ontologies located on the node as well as about their underlying data repositories. Its main task is to answer queries formulated over an ontology for the retrieval of data from the repositories. In order to achieve this task, the Ontology Server utilizes mappings between terms in the ontology and data structures in the underlying repositories as well as the appropriate wrappers.

A *wrapper* is a module which understands a specific data organization. It knows how to retrieve data from the underlying repository and hide the specific data organization to the rest of the system. With Internet sites as data repositories, a wrapper emulates the behavior of a user, who is accessing data from a Web site.

## 2.3 Discussion

In the first section of this chapter, we presented the main issues in information integration using ontologies. These issues include: the type of underlying information sources, the architecture type, the use of ontologies, the ontology representation language and the relationship between global and local schemas. In the previous section, we reviewed several ontology-based information integration systems. Reflecting the presented systems on the issues presented in Section 2.1, we obtain the Table 2.1.

From this survey we learn several lessons. First of all, an integration system should support the main types of information sources used in modern information systems. In particular, relational databases and XML data sources are nowadays the dominant means to store information.

Integration systems should also be dynamic and take in account the addition and removal of information sources. Dynamic integration systems should provide some means to know which sources are available at a given moment.

However, the best way to support the instability of information sources is the use of hybrid ontology approach. We have seen that in the hybrid ontology approach each information source is described by its own ontology and those local ontologies are built from a global shared vocabulary. Thus, the advantage is that, on the one hand, new sources can easily be added without the need of modification. On the other hand, local ontologies become comparable to each other.

Regarding the architecture type, we can observe that only few systems use agent-based architectures (KRAFT and InfoSleuth). This might be because the complexity of such architectures, where the communication among agents is expensive to implement and the agent interaction is difficult to understand.

We can also observe that most of integration systems adopt the wrapper/mediator architecture (BUSTER, COIN, MOMIS, etc.). In this type of architecture, the mediator is used to create an integrated view of data over multiple sources, and the wrapper is used to map local information sources to a common data model. Indeed, a mediation-based architecture is more comprehensible and easier to implement than an agent-based architecture.

We think that a combination of the hybrid ontology approach with the wrapper/mediator architecture makes the best enhancement of ontology-based integration systems. Such a combination can be applied when the shared vocabulary of the hybrid approach is a global ontology that represents an integrated/mediated view over multiple local ontologies. Reviewing Table 2.1 tells us that the only systems which combine hybrid ontology approach and wrapper/mediator architecture are BUSTER and COIN.

However, another enhancement can also be obtained by considering local ontologies at site level rather than at source level. That is, a local ontology represents the semantics of a set of information sources located at a physically or logically separated site. In other words, instead of wrapping each single local information source to its own local ontology, a cluster of information sources are wrapped to a single local ontology. Different local ontologies are always related to the global ontology. In some sense, this consideration means that a single ontology approach is also used at the local/site level. We can affirm that no one of cited systems take this point in account.

In fact, our vision about an ontology-based integration system is characterized by two main features:

1. Combining the hybrid ontology approach with the mediation-based architecture at the global level.
2. Using a single ontology approach at the local/site level.

Before explaining our vision, we illustrate what we mean by "site". A site is a set of physically or logically related information sources. We say that two information sources are physically related if they are located at the same physical location i.e. same machine or same network node. We say that two information sources are logically related if they share the same domain of interest (they can be physically separated).

Now, our vision means that at each site, underlying information sources are wrapped to a single local ontology (site level), and different local ontologies of different sites are mapped to a global ontology that represents the whole domain of interest (global level).

At the site level, mappings are needed between underlying sources and the local ontology. Constructing these mappings depends on the nature of the local ontology. In fact, if the local ontology is created from information sources, then constructing mappings can be done automatically. However, if the local ontology exists independently of information sources, then mappings can be constructed manually or semi-automatically.

At the global level, mappings are also needed between local ontologies and the global ontology. Since the global ontology is obviously independent of local ontologies, mapping construction can be done semi-automatically.

In our vision, information sources are transparent to end users. That is, users can query the integration system without knowing the location, structure, data model and the availability of underlying information sources. To achieve this, users submit their queries in terms of the global ontology. User queries are firstly decomposed and translated over local ontologies. Queries over local ontologies are then translated over local information sources.

An important question arises here: regarding GAV and LAV approaches presented in Section 2.1.5, which approach is appropriate to our vision. To answer this question, we distinguish two levels of integration:

1. *Site level* – At this level, local information sources are related to a local ontology. That is, the local ontology represents a global schema $g$ over local sources $s$. We consider that at each site, the underlying information sources are stable and do not usually change. Therefore, the appropriate approach to use at this level is GAV approach. This means that the local ontology (global schema) is expressed in terms of local information sources.
2. *Global level* – At this level, local ontologies are related to the global ontology. That is, the global ontology represents a global schema $g$ over local ontologies $s$ (sources at this level). We consider that the global ontology is stable and immune to change. Whereas, local ontologies can be added to or removed from (connect or disconnect) the integration system. Therefore, the appropriate approach for this level is the LAV approach. This means that local ontologies are defined as views over (in terms of) the global ontology.

Figure 2.4 illustrates our vision where GAV approach is applied at the site level, and LAV approach is applied at the global level.

The last thing to mention in this discussion concerns the ontology specification language. An ontology-based integration system should make advantage of modern ontology languages that combine the expressiveness and reasoning power. The Web Ontology Language (OWL) is one of the best current ontology languages and becomes a W3C recommendation (Section A.5).

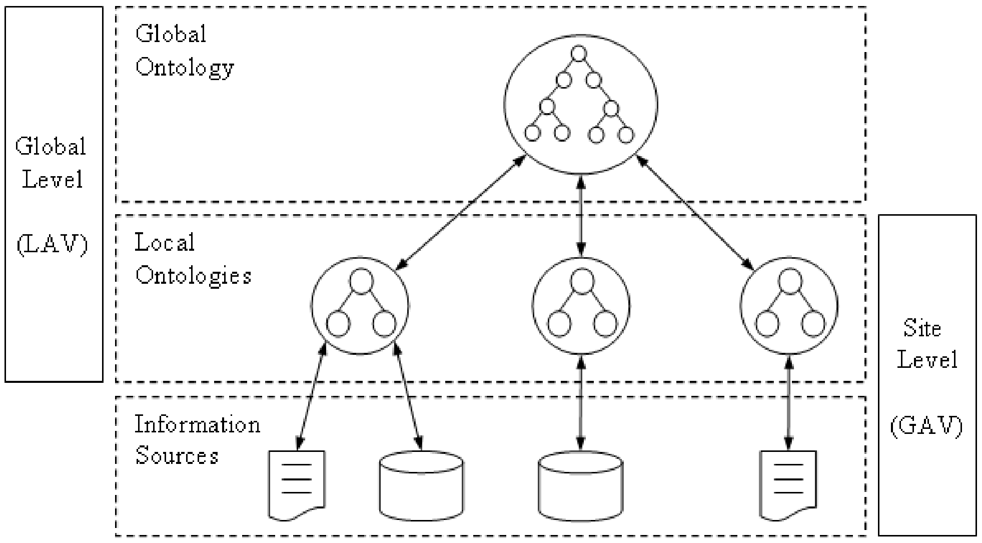


FIGURE 2.4: Combining GAV and LAV approaches in Hybrid Ontology Approach

## Chapter Summary

In this chapter, we have given a background on information integration systems that are based on ontologies. We firstly reviewed the main issues that concern ontology-based information integration systems. These issues include: the type of underlying information sources, the architecture type, the use of ontologies, the ontology representation language and the relationship between global and local schemas.

We have also presented several ontology-based information integration systems which considered well-known in the literature. Presented systems are: BUSTER, COIN, DOME, InfoSleuth, KRAFT, MOMIS, and OBSERVER. Table 2.1 reflects these systems with respect to the considered issues.

A discussion about the current state of the art has led us to introduce our vision about an enhanced ontology-based information integration system. We have argued that such a system should take several points into account, including:

1. support both relational databases and XML data sources,
2. support dynamic addition and removal of information sources,
3. adopt mediation-based architecture,
4. adopt hybrid ontology approach,
5. consider local ontologies at site level (vs. source level),
6. use GAV approach at site level, and LAV approach at global level, and
7. use OWL ontology language.

In the next chapter, we will introduce our proposed ontology-based information integration system that fulfils all these requirements and, thus, realize our vision.

TABLE 2.1: Main features of ontology-based information integration systems

| | **Ontology Use** | **Architecture Type** | **Data Model** | **Level of Mapping** | **Degree of Automation** | **Query Language** | **GAV / LAV** |
|---|---|---|---|---|---|---|---|
| **BUSTER** | Hybrid | wrapper/mediator | XML, RDF, OWL | ontology mapping, semantic translation | semi | SQL like, QXML, QTDF | – |
| **COIN** | Hybrid | wrapper/mediator | Relational, F-Logic | schema mapping | semi | SQL, global | GAV |
| **DOME** | Multiple | wrappers, mapping server, ontology server | CLASSIC | ontology mapping | manual | SQL like | – |
| **InfoSluth** | Multiple | agent-based | KQML, LDL | schema & ontology mapping by Agents | semi | KQML/KIF | – |
| **KRAFT** | Hybrid | agent-based | frame-based | – | semi | CIF, CoLan | – |
| **MOMIS** | Single | wrapper/mediator | $ODL_{I^3}$ | schema integration | semi | $OQL_{I^3}$ | GAV |
| **OBSERVER** | Multiple | wrappers, ontology servers, IRM | CLASSIC, LOOM | ontology mapping & translation | manual | CLASSIC | – |

# Chapter 3

# OWSCIS Overview

Contents

## Abstract

In Chapter 2, we discussed existing ontology-based information integration systems and their features. This discussion has led us to introduce, in Section 2.3, our vision about an enhanced ontology-based information integration system. We argued that such a system should:

1. support both relational databases and XML data sources,
2. support dynamic addition and removal of information sources,
3. adopt mediation-based architecture,
4. adopt hybrid ontology approach,
5. consider local ontologies at site level (vs. source level),
6. use Global-As-View approach at site level, and Local-As-View approach at global level, and
7. use OWL ontology language.

In order to realize our vision, we propose an ontology-based information integration system that fulfils all these requirements. Our proposed system is called OWSCIS, which is an acronym of: ***Ontology and Web Service based Cooperation of Information Sources***. In Section 3.1, we give a general overview of OWSCIS system. A more detailed description of OWSCIS components is given in the reminder of this chapter.

## 3.1 OWSCIS Architecture Overview

OWSCIS is an ontology-based cooperation system between distributed heterogeneous information sources [75][72]. Its purpose is to allow users to query information sources simultaneously and to combine obtained results in order to transparently get information that may not be available directly.

OWSCIS is based on ontologies to represent the semantics of information sources that can be structured, such as relational databases, and/or semi-structured, such as XML documents. OWSCIS is intended to support dynamic/instable information sources. That is, sources can connect to or disconnect from the cooperation platform at any time.

OWSCIS architecture belongs to the hybrid ontology approach, that is, local information sources are mapped to local ontologies while a global ontology is used as a global model for all participating local ontologies. The advantage of wrapping local information sources to a local ontology is to preserve the autonomy of participating sites and the transparency of underlying local information sources. Thus, this will simplify the integration task by supporting dynamic addition and removal of participating sites.

In fact, information sources are normally distributed over multiple separated sites. A site may contain several information sources. At each site, a local ontology is used to describe the semantics of local information sources. In other words, a local ontology represents a pivot data model that unifies the structures of all information sources within a site.

This local ontology has to be linked to real information. Therefore, mappings have to be provided between the local ontology and each information source. Moreover, at the site level, a GAV approach is used to relate information sources to their local ontology (which plays the role of global schema at this level). This means that the local ontology is expressed (as virtual view) in terms of local information sources.

Beside local source ontologies, a global centralized ontology is used to describe the semantics of the whole domain of interest. Different local ontologies of different sites are related to this global ontology. In other words, this global ontology is a reference for all incorporated local ontologies and it is supposed to cover their domains. In order to interconnect local ontologies, they are mapped to the global ontology which plays the role of mediator. All the specified concepts of the local ontologies are assumed to have similar concepts in the global ontology. Each local ontology refers to specific parts of a domain which is globally covered by a reference ontology.

Moreover, at the global level, a LAV approach is used to relate local ontologies (which play the role of sources) to the global ontology (which plays the role of global schema at this level). This means that local ontologies are expressed as virtual views in terms of the global ontology.

**Example** – Figure 3.1 shows an enterprise that is distributed over four sites: a headquarters and three branches. Each site includes one or more information source, that can be relational databases or XML data sources. At each site, local information sources are mapped to a single local ontology. Different local ontologies of different sites are related to the global centralized ontology which is located at the enterprise headquarters.

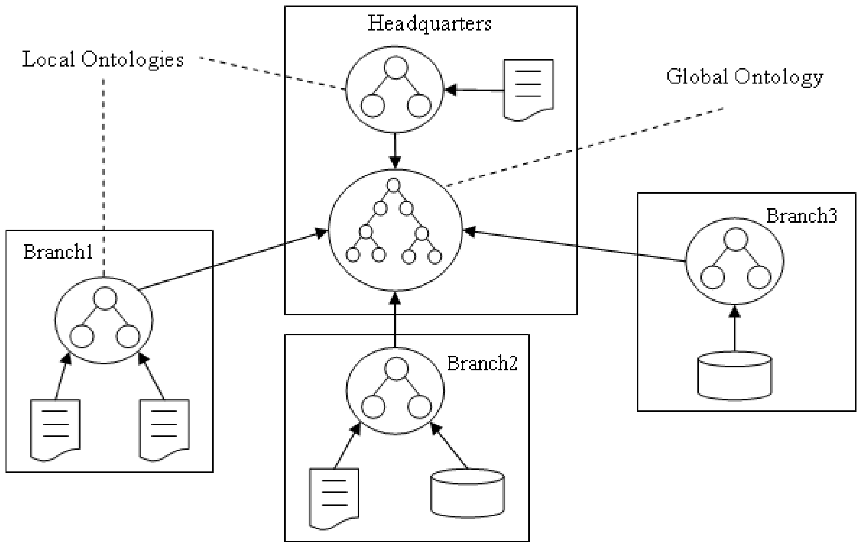


FIGURE 3.1: Local and global ontologies within a distributed enterprise.

Local ontologies and the global ontology are both described in OWL language, which is a W3C recommendation for publishing and sharing ontologies on the web (Section A.5). More precisely, we use OWL-DL sublanguage because it is based on Description Logics [7], and is characterized by its expressiveness and reasoning power.

OWSCIS system uses a non-materialized approach. That is, data instances are not materialized neither at local ontologies nor at the global ontology. Local and global ontologies only contain concepts and properties, but not the instances. A query-driven approach is used to populate ontological instances with data coming from the information sources. That is, instances are not stored in the ontologies but retrieved on the fly as response to user queries.

OWSCIS architecture deals with all steps needed to interconnect various information sources. These steps include:

- creation of local ontologies
- mapping information sources to local ontologies
- mapping local ontologies to the global ontology
- query processing: decomposition, rewriting, translation, reformulating, and recomposition, and
- visualization of ontologies and query results.

OWSCIS architecture consists of several modules and web services, each of them aims at performing a specific task (Figure 3.2):

1. **Knowledge Base Module** – it is a unique mediator used to encapsulate the global ontology with a toolbox and a mapping directory.
2. **Data Provider** – it is a wrapper that associates local information sources, at a specific site, with their local ontology.
3. **Mapping Web Service** – it is used to establish mappings between the local ontologies and the global one.
4. **Querying Web Service** – it is used to process user queries, i.e. analyze the queries, decompose them into sub-queries which are redelivered to the relevant data providers, and recompose incoming results.
5. **Visualization Web Service** – it is used to visualize the global ontology and suitably present query results to the end user.

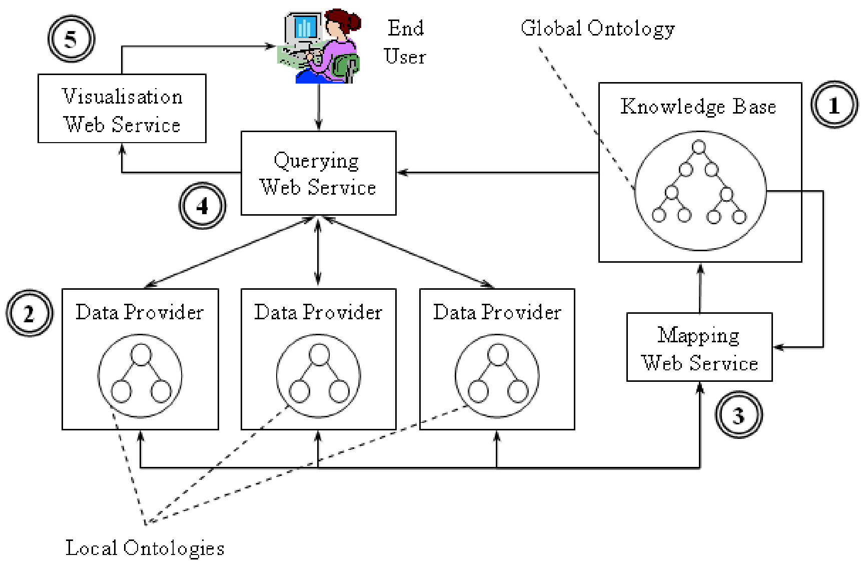


FIGURE 3.2: The components of OWSCIS architecture.

Figure 3.3 demonstrates how the different components of OWSCIS can be deployed to the enterprise presented in the previous example. In the reminder of this chapter, we describe in more details the different components of OWSCIS system.

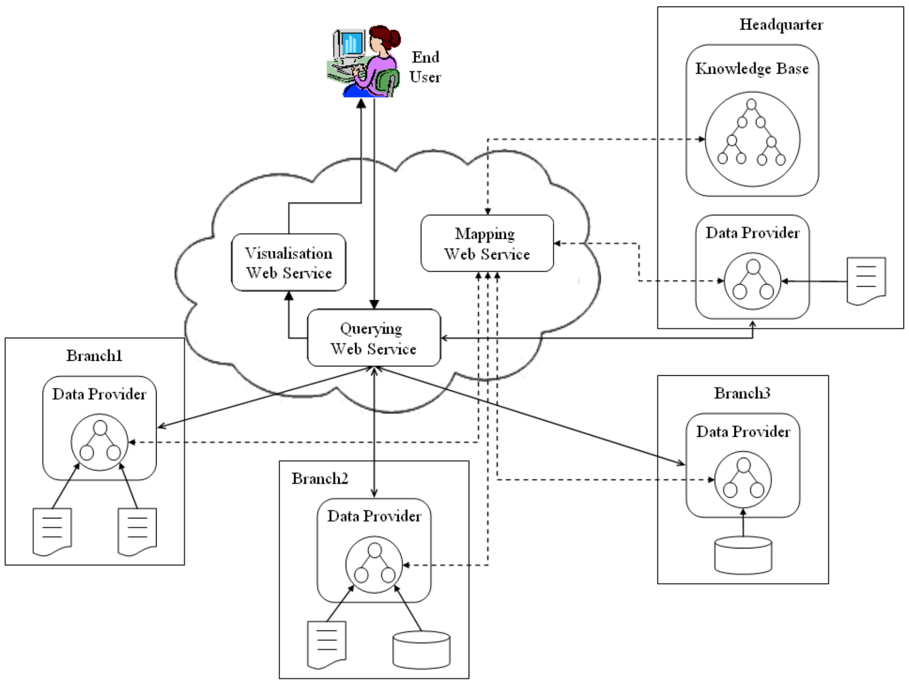


FIGURE 3.3: OWSCIS architecture applied onto a distributed enterprise.

## 3.2 Knowledge Base Module

This is the main component of the architecture, it centralizes all information needed to exploit the whole cooperation system. It is composed of a global ontology, a mappings directory, and a toolbox (see Figure 3.4).

The global ontology describes the semantics of a given domain of interest. This description is assumed to be complete, i.e, it includes all concepts related to the described domain. The global ontology also represents a global model for local ontologies, therefore, it covers all the local domains, i.e. each concept (resp. property) in any local ontology should have a corresponding concept (resp. property) in the global ontology. As mentioned early in this chapter, the global ontology is described in OWL-DL Language.

The mapping directory is a means to provide information about available data providers, and about mappings between the global ontology and local ontologies. The mapping directory is a simple table listing the concepts from the global ontology and associating each concept with a list of available data providers that have an equivalent concept. The mapping directory is used to quickly find which data providers are relevant to a given query. It does not contain the exact mappings which are stored in the various data providers. The mapping directory should be updated during the mapping process.

The toolbox contains all information needed to perform the mapping between local ontologies and the global ontology. It includes a set of tools and methods that are used by the mapping web service to estimate the similarity between ontologies components. It describes the way the mapping estimation will happen. It basically list which methods should be used for the similarity estimation and their relative importance [134]. If several mapping methods are available, it also defines which one, or which combination of them has to be used.

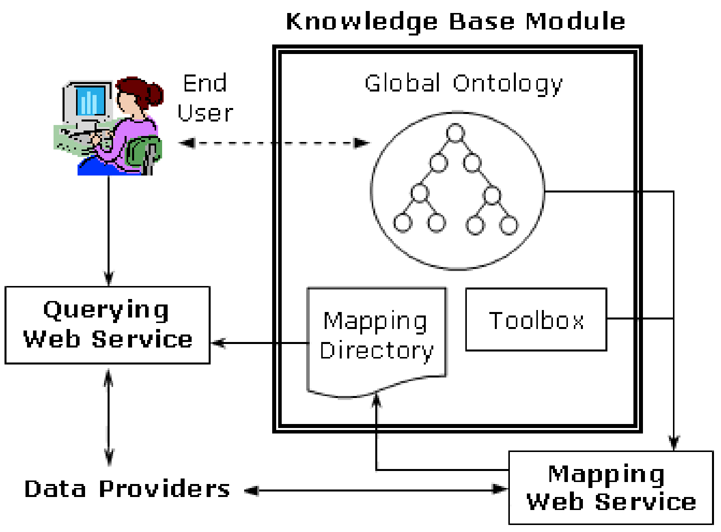


Figure 3.4: Knowledge Base Module

## 3.3 Data Provider

A participant site in OWSCIS cooperation system can contain several information sources of different types. In particular, relational databases and XML data sources are supported. A data provider is a module that encapsulates the set of information sources located at a single site. The main purpose of a data provider is to wrap local information sources within a site to a local ontology.

A data provider contains the following components (Figure 3.5):

- *Local ontology* – represents a unified data model of local information sources.
- *Mappings* between the local and the global ontologies
- *Mappings* between the local ontology and local information sources
- *Set of mapping tools* – serve to map information sources to the local ontology and process local queries. (*DB2OWL* and *X2OWl*)

Using these components, a data provider performs three fundamental tasks:

1. Create the local ontology.
2. Map the information sources to the local ontology.
3. Process local queries.

### 1. Create the local ontology

The local ontology of the data provider does not necessarily exist before the connection to the platform. In this case, the local ontology has to be created from the information sources. This task is handled using the tools: 1) DB2OWL [51][72] for a single relational database, and 2) X2OWL [74] for a single XML data source. For the case of multiple relational databases and XML data sources, we conducted some works such as [73], but we did not yet implement a tool.

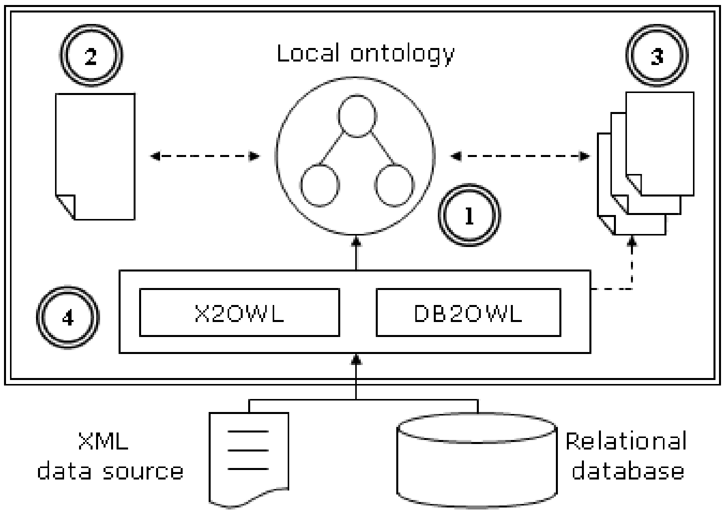


FIGURE 3.5: Data Provider

Since we use a non-materialized approach, the generated ontology only contains the concepts and properties but not the instances. Data instances are retrieved and translated as needed in response to user queries. During ontology generation process, these tools also generate a mapping document for each information source. This mapping document describes the correspondences between the information source and the generated local ontology. The mapping document is used for query processing purposes.

## 2. Map the information sources to the local ontology

A participating site may already have its own local ontology, and do not want to create a new one from the information sources. In this case, each local information source can be mapped to the existing local ontology. This task is handled using DB2OWL for a relational database, and X2OWL for an XML data source. The result of this process is a mapping document that describes the correspondences between the information source and the local ontology.

## 3. Process local queries

A data provider is intended to handle all query processing steps needed to solve local queries. These steps include: query rewriting, query translation, and results reformulation.

**Query rewriting** – When a query is received by a data provider, it is firstly rewritten into a query in terms of the local ontology. This rewriting phase is based on the mappings between the local ontology and the global ontology.

**Query translation** – The rewritten query is then translated into queries over local information sources (in SQL for relational databases, and in XQuery for XML data sources). This translation phase is based on the mappings between the local ontology and underlying information sources.

**Result reformulation** – After resolving queries in the underlying sources, the results are reformulated according to the local ontology. These query processing tasks are performed using DB2OWL and X2OWL tools.

## 3.4 Mapping Web Service

The purpose of this service is to map local ontologies to the global ontology. In order to add a new data provider to OWSCIS platform, its local ontology must be mapped to the global ontology using the mapping web service. The mapping web service compares the two ontologies using the methods defined inside the toolbox of the knowledge base module, and produces inter-ontology mappings which will be stored into the relevant data provider. The mapping process also updates mappings directory in the knowledge base module.

The mapping process consists of four main steps: preprocessing, similarity estimation, refining and exploitation (Figure 3.6). The purpose of the preprocessing step is to clean up and prepare the data for the next phases.

**Similarity Estimation** – In the similarity estimation phase, the mapping web service estimates the similarity between the components (concepts and properties) of the two ontologies, using a combination of structural and semantic methods [132][134][131]. Structural methods estimate structural similarity by comparing concept names as a string using distance edition techniques. Semantic methods extract known words from the concepts names and perform a semantic similarity estimation over them. The mapping web service rely on an external resource in order to estimate the semantic similarity. WordNet thesaurus [61] is used for this purpose. The result of the similarity estimation phase is a numerical similarity estimation value for each pair of concepts or properties.

**Refining** – The purpose of this phase is to solve cases where the similarity value between two concepts is insufficient to determine whether they are equivalent or not. The similarity estimation value between two concepts is refined according to the structure of their neighborhood in compared ontologies.

**Exploitation** – In this phase, similarities are translated from their numerical values into proposals of mappings, and the overall mapping between the two ontologies is automatically produced [133]. The proposals of mappings can be then validated and modified by experts. Once the mappings are validated, they are stored locally in the pertinent data provider and the mappings directory is suitably updated.

## 3.5 Querying Web Service

Once the various data providers are connected to the knowledge base module, users can query them using the querying web service. When a query (expressed in SPARQL language [138]) is submitted to the system, it is analyzed by this service and decomposed into a set of sub-queries. Then, using the mapping directory in the knowledge base, the query web service redirects the sub-queries to the suitable data providers (Figure 3.7).

As mentioned in Section 3.3, when a query is received by a data provider, it is firstly rewritten into a query in terms of the local ontology (in SPARQL). This rewriting phase is based on the mappings between the local ontology and the global ontology. The rewritten query is then translated into one or more queries over local information sources (in SQL for relational databases, and in XQuery for XML data sources). This translation phase is based on the mappings between the local ontology and underlying information sources. The translated queries are solved

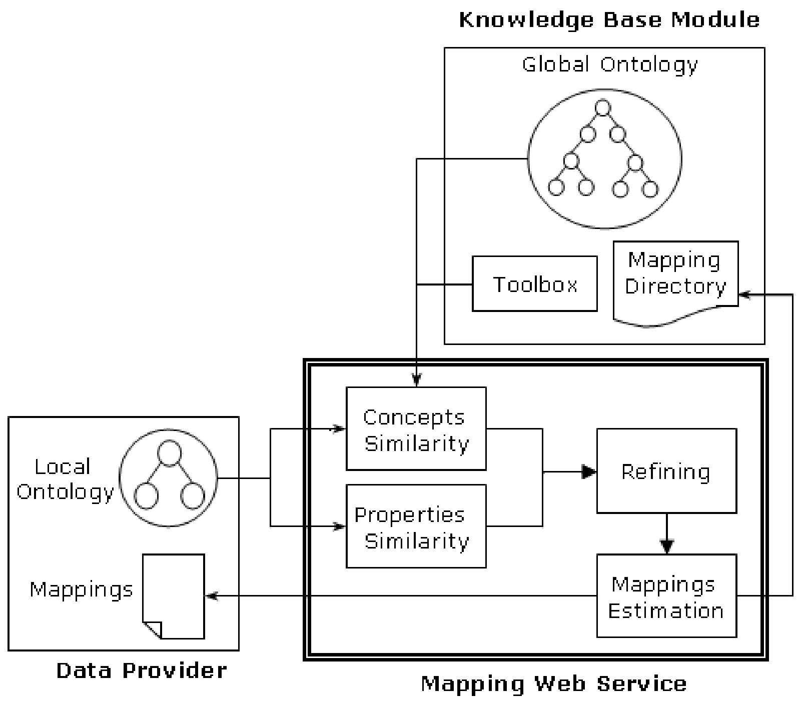


FIGURE 3.6: Mapping Web Service

in the underlying sources, and their results are reformulated according to the local ontology. The reformulated results are returned to the querying web service.

The query web service collects the responses returned from data providers and recomposes them in one coherent response which is sent to the visualization web service. The final answer is redirected to the visualization web service which displays it in an easily-understandable way for the user.

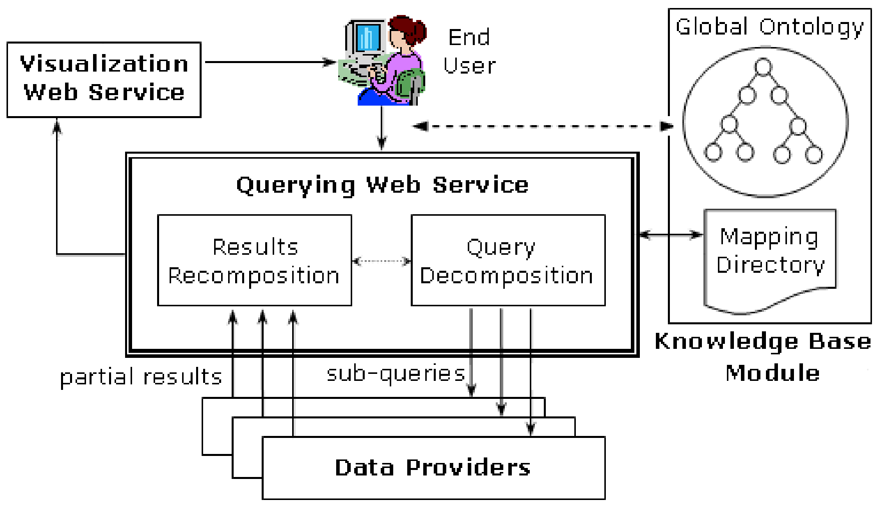


FIGURE 3.7: Querying Web Service

## 3.6 Visualizing Web Service

The visualization service proposes different functionalities including the visualization of the global ontology and the visualization of the queries and their results [1]. It offers a graphical interface allowing the navigation through the global ontology and visualizing the concept hierarchy and the properties with their domains and ranges.

The visualization service is also used to show the results of a query in a dynamic and user-friendly way. The query results are viewed as instances of the concepts and properties specified in the query and satisfying its restrictions [79].

## Chapter Summary

In this chapter, we have introduced OWSCIS system, our contribution to information integration using ontologies. OWSCIS is an ontology-based cooperation system between distributed heterogeneous information sources. OWSCIS architecture belongs to the hybrid ontology approach, that is, local information sources are mapped to local ontologies while a global ontology is used as a global model for all participating local ontologies. The relationship between local ontologies and the global ontology follows LAV approach. Moreover, a GAV approach is used to relate information sources to their local ontology.

OWSCIS architecture supports all steps needed to interoperate various information sources, including: 1) creation of local ontologies, 2) mapping information sources to local ontologies, 3) mapping local ontologies to the global ontology, 4) query processing, and 5) visualization of ontologies and query results.

We have also given an overview on the different components of OWSCIS, including: Knowledge Base Module, Data Provider, Mapping Web Service, Querying Web Service, and Visualization Web Service.

In the next chapter, we will emphasize the principal contributions of this dissertation within OWSCIS framework. These contributions include:

- A method for creating an ontology from a relational database.
- A method for mapping a relational database to an existing ontology.
- A method for translating SPARQL to SQL queries.
- A method for creating an ontology from an XML data source.
- A method for translating SPARQL to XQuery queries.
- A method for creating an ontology from multiple information sources.

Other aspects of OWSCIS framework, such as the inter-ontology mapping process [133] [134][132][131], and the visualization of ontologies and query results [1] [79] are the subject of other PhD thesis works of my colleagues: Thibault Poulain and Guillermo Valente Gómez Carpio.

# Chapter 4

# Contributions Overview

**Contents**

## 4.1 Introduction

In the previous chapter, we introduced OWSCIS system, our ontology-based cooperation system and we gave an overview of its main components. We saw that OWSCIS architecture consists of a knowledge base module, a set of data providers, and a set of web services (namely, mapping, querying and visualization web services).

In this chapter, we emphasize the principal contributions held in this dissertation within OWSCIS framework. In fact, in this dissertation, we focus our interest on a major issue which is the mapping of information sources (relational databases and XML data sources) to local ontologies. Hence, the concerned component of OWSCIS is the data provider.

We saw in Section 3.3 that the data provider performs three fundamental tasks:

1. Create the local ontology from local information sources.
2. Map the local information sources to the local ontology (if already exists).
3. Translate queries over the local ontology into queries over the local information sources.

We also distinguish three cases according to number and type of the information sources contained in a site:

- The site contains a single database.
- The site contains a single XML data source.
- The site contains multiple databases and/or XML data sources.

Thus, for each of these three cases, we have to propose a method for handling the three mentioned tasks, totally giving nine problems to solve. Fortunately, we treat seven of these nine problems, as shown in the following table:

| | ontology creation | mapping to existing ontology | query translation |
|---|---|---|---|
| single database | yes (chapter 7) | yes (chapter 7) | yes (chapter 8) |
| single XML data source | yes (chapter 10) | no | yes (chapter 11) |
| multiple databases and XML data sources | yes | yes | no |

However, due to lack of space, in this dissertation we only present our works about the cases of single database and single XML data source, whereas our works about the case of multiple databases and XML data sources are not presented in this dissertation. We refer to our paper [73] as reference on these absent works.

In the following sections, we give a brief overview on these contributions.

## 4.2 Database-to-Ontology Mapping

### 4.2.1 Database-to-ontology Mapping Specification

In Chapter 6, we propose two database-to-ontology mapping specifications, namely:

- **Associations with SQL statements** – In this kind of mappings, ontology terms (concepts and properties) are associated with simple SQL statements over the database (Section 6.2).
- **DOML language** – This language is adapted from existing database-to-ontology mapping languages (D2RQ, R2O and Relational.OWL) and is based on RDF. It allows the specification of sophisticated database-to-ontology mappings in the form of semantic bridges expressed as RDF statements (Section 6.3).

### 4.2.2 Ontology Creation from a Relational Database

We propose a method for ontology creation from a single database. This method is implemented as a tool called DB2OWL which is presented in Chapter 7.

This ontology creation method (7.3) relies on detecting some particular cases for tables in the database schema. According to these cases, each database component (table, column, constraint) is then converted to a corresponding ontology component (class, datatype property, object property).

During ontology generation process, DB2OWL also generates a mapping document that describes the correspondences between the database and the generated local ontology. The mapping document is used for query processing purposes. Thus, we also present two methods for generating the mappings for our two proposed mapping specifications: associations with SQL statements (Section 7.3.5.1) and DOML (Section 7.3.5.2).

### 4.2.3 Mapping a Relational Database to an Existing Ontology

The DB2OWL tool is also able to map a database to an existing ontology. It provides a graphical interface that allows a human expert to manually indicate the mappings between the components of a database and an existing ontology. This mapping process does not require changing neither the ontology nor the database.

The result of this process is a mapping document that describes the correspondences between the database and the ontology. This mapping document can be expressed using one of the two proposed kinds of mapping specifications: Associations with SQL statements or DOML.

Section 7.4 presents this capability in details.

### 4.2.4 SPARQL-to-SQL Query Translation

In Chapter 8, we treat the problem of translating queries over the local ontology (in SPARQL) into queries over the local database (in SQL). We propose two translation methods for the two proposed mapping specifications: associations with SQL statements (Section 8.3), and DOML (Section 8.4).

The translation method using associations with SQL statements consists of three steps:

1. SPARQL query parsing.

2. SPARQL to SQL query translation – treats the extracted query components and the BGP with the mapping document, and constructs the components of the target SQL query (SELECT, FROM, and WHERE clauses).
3. Simplification of the translated SQL query – aims at rewriting the translated SQL query into a simplified query which is more readable to users and reduces the number of relational operations and avoids unnecessary conditions.

The translation method using DOML language consists of four steps:

1. DOML mapping document parsing.
2. SPARQL query parsing.
3. Preprocessing – aims at computing an auxiliary mapping between the variables of the SPARQL query and the concepts of the ontology.
4. SPARQL to SQL query translation –constructs the components of the target SQL query (SELECT, FROM, and WHERE clauses).

## 4.3 XML-to-Ontology Mapping

### 4.3.1 XML-to-ontology Mapping Specification

In Chapter 10, we propose an XML-to-ontology mapping specification, called XOML. This language is based on RDF and similar to DOML language. It allows the specification of sophisticated XML-to-ontology mappings in the form of semantic bridges expressed as RDF statements. This language is presented in Section 10.3.

### 4.3.2 Ontology Creation from an XML Data Source

We propose a method for ontology creation from a single XML data source. This method is implemented as a tool called X2OWL which is presented in Chapter 10. X2OWL can automatically generate an OWL ontology from an XML schema. This process is based on some mapping rules that indicate how to convert each component of the XML schema to a semantically corresponding ontology component. During ontology generation process, X2OWL also generates a mapping document that describes the correspondences between the XML data source and the generated local ontology. The mapping document is expressed using the proposed mapping specifications: XOML.

### 4.3.3 SPARQL-to-XQuery Query Translation

In Chapter 11, we treat the problem of translating queries over the local ontology (in SPARQL) into queries over the local XML data source (in XQuery). We propose a translation method for the proposed mapping specification XOML. This translation method consists of four steps:

1. Parsing the XOML mapping document.
2. Parsing the SPARQL query.

3. Preprocessing – the purpose of this step is to compute all possible mapping graphs corresponding to the basic graph of the SPARQL query.
4. Building the XQuery query – this is the main step, where the different clauses of the XQuery query (for, let, order by, where, and return) are constructed.

# Chapter 5

# Background on Database-to-Ontology Mapping

**Contents**

## Abstract

In this chapter, we will introduce the field of database-to-ontology mapping. We firstly give a background on the main issues addressed in the topic of database-to-ontology mapping, including the context of usage, the nature of the target ontology, the mapping definition and the data migration, etc. Then, we review the literature to investigate some of existing works in this field. In particular, we are interested in database-to-ontology mapping specifications and approaches. Finally we will discuss the presented issues and approaches.

## 5.1 Issues in Database-to-Ontology Mapping

Database-to-ontology mapping is the process whereby a database[1] and an ontology are semantically related at a conceptual level. The result of this process is a set of correspondences (mapping elements) that relate the components of a database (tables, columns, primary and foreign keys, etc.) with the components of an ontology (concepts, relations, attributes, etc.).

We should draw the attention of the reader to that our coverage of the topic of database-to-ontology mapping does not include tools or frameworks that allow simple storage of ontology data in a relational database, such as: 3Store [88], Sesame [36], Instance Store [91], and DLDB [130]. In these tools, the database is used to store ontological data but the user does not define mappings (he can only interact with the ontology data layer). Such tools are out of the scope of this dissertation.

In this section, we present the main issues that characterize database-to-ontology mapping approaches. These issues include: the context within which the mapping approach is used, the nature of the target ontology, the mapping definition, the instance migration, the automation level, and the querying process.

### 5.1.1 Usage Context

The first question one may ask is: why we need to map databases to ontologies?. In fact, there are some contexts within which database-to-ontology mapping is needed. We retain two contexts that are important to our work. Firstly, in ontology-based information integration (see Chapter 2) where ontologies are used to solve semantic heterogeneity, it is necessary to relate local sources (typically, databases) to conceptual models (ontologies). The second context is the Semantic web, where ontologies are used to annotate deep web sources.

- **Ontology-based Information Integration**

  As we saw in Chapter 1, semantic interoperability is a key application of ontologies. In order to overcome the problem of semantic heterogeneity of information sources, ontologies are usually used for the explication of implicit and hidden knowledge in information sources involved in the integration. In this context, the database-to-ontology mappings are essential to relate the ontologies to the actual content of information sources. Many ontology-based information integration approaches have been developed. In Section 2.2,

[1]In the reminder of this dissertation, we will use the term "database" to briefly denote a "relational database".

we presented a survey about ontology-based information integration systems. In such systems, mappings are usually exploited to relate ontologies to information sources.

- **Semantic Web**

  The semantic web is, as originally proposed by Tim Berners Lee et al. [21], an extension of the current web, in which the web content can be expressed in a way that, in addition to be human readable, can be understood by software agents. Recently, ontologies have become a cornerstone of the Semantic Web, which has the model of distributed, reusable, and extendable ontologies at its core.

  There is a large quantity of data on Web pages generated from relational databases. This information is often referred to as the Deep Web [19] as opposed to the surface Web comprising all static Web pages. Ontologies are used to upgrade this large amount of existing content into Semantic Web content which are expressed in formal specifications instead of natural language. Tim Berners Lee affirms that "*one of the main driving forces for the Semantic web, has always been the expression, on the Web, of the vast amount of relational database information in a way that can be processed by machines*" [20].

  The way of adding semantics to the dynamic web pages is the use of meta annotations which publish database content, with information about the underlying database and how each content item in a page is extracted from the database. This process is known as "*Annotation of Deep Web*". A good presentation to this usage is given by Volz et al. in [160].

### 5.1.2 Target Ontology

According to the nature of the target ontology (to which the database is mapped), we may distinguish two main categories of database-to-ontology mapping approaches: 1) approaches that create an ontology from a database, and 2) approaches that map a database to an existing ontology.

- **Ontology creation from a database** – The objective of these approaches is to create an ontology model from a database model and to migrate the contents of the database to the generated ontology. The mappings here are simply the correspondences between each created ontological component (concept, property, . . . ) and its original database component (table, column, . . . ).

- **Mapping a database to an existing ontology** – In these approaches, an ontology and a legacy database already exist and the goal is to create mapping between them, and/or populate the ontology by the database content.

Both mapping approaches above include two processes: 1) mapping definition i.e., the transformation of database schema into ontology structure, and 2) data migration i.e., the migration of database contents into ontology instances. These two processes are presented below.

### 5.1.3 Mapping Definition

Before thinking about how to define mappings from databases to ontologies, we should think about what we can map, and what we cannot map.

Databases and ontologies have heterogeneous expressiveness capabilities. Relational databases have a set of static constructs including: tables, columns, primary/foreign keys, datatypes, and integrity constraints, as well as a variety of behavioral features (such as triggers, stored procedures, referential actions etc.). Due to the static nature of ontologies, only the static part of relational databases can be mapped to ontologies, whereas dynamic aspects of relational databases (such as triggers) cannot be mapped [148].

The definition of database-to-ontology mappings varies according to the nature of the target ontology, that is, whether it is created from the database or it already exists independently from the database:

- When an ontology is created from a database (first category), the database model and the generated ontology are very similar. Therefore, mappings are quite direct, and complex mapping situations do not usually appear. However, the creation of ontology structure may be done in two possible ways:
  - **Simple** – involving straightforward transformation of every database table into an ontology concept and every column into a property. This type of direct mapping is not enough for expressing the full semantics of the database domain.
  - **Complex** – involving the discovery of additional semantic relations between database components (like the referential constraints) and take them into account while constructing ontology concepts and relations between them.
- When a database is mapped to an existing ontology (second category), mappings are more complex than those in the previous case because the modelling criteria used for designing databases are different from those used for designing ontology models. Complex mapping situations usually arise from low similarity between the ontology and the database model (one of them is richer, more generic or specific, better structured, etc., than the other). Moreover, the ontology domain and the database domain do not always coincide.

Barrasa-Rodriguez et al. present in [8] three different levels of overlap between the domains covered by the database and the ontology (see Figure 5.1):

1. **Partial Intersection** – In this level ontology domain $\mathcal{O}$ and the database domain $\mathcal{D}$ partially intersected, i.e., a subset of $\mathcal{D}$ entities corresponds to a subset of $\mathcal{O}$ entities:
$$\mathcal{O} \cap \mathcal{D} \neq \phi$$
$$\mathcal{O} \backslash \mathcal{D} \neq \phi$$
$$\mathcal{D} \backslash \mathcal{O} \neq \phi$$
2. **Inclusion** – In this level one of the domains is included in the other. The ontology domain $\mathcal{O}$ can be included in the database domain $\mathcal{D}$:
$$\mathcal{O} \cap \mathcal{D} = \mathcal{O}$$
or, the database domain $\mathcal{D}$ is included in the ontology domain $\mathcal{O}$:
$$\mathcal{O} \cap \mathcal{D} = \mathcal{D}$$

3. **Coincidence** – In this level the ontology domain $O$ and the database domain $D$ coincide.
$$\mathcal{O} = \mathcal{D} = \mathcal{O} \cap \mathcal{D}$$
This level can be typically found when the ontology is created from the database.

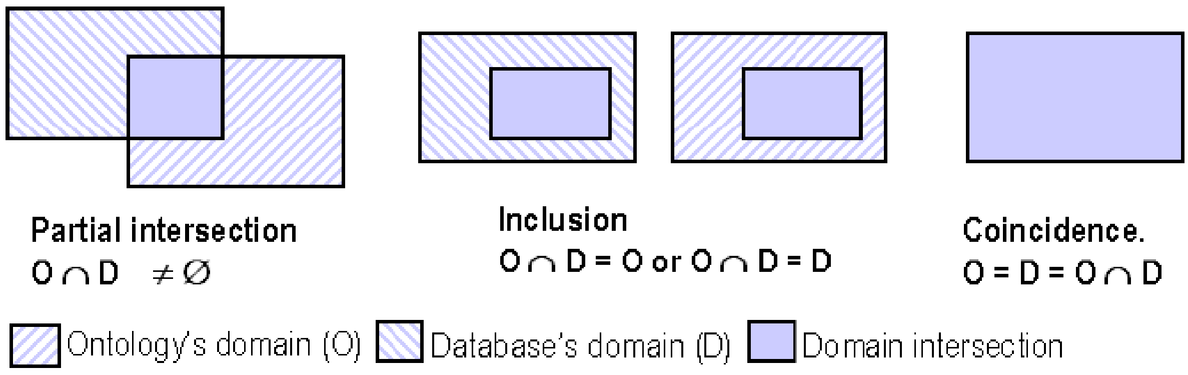


FIGURE 5.1: Levels of overlap between the domains of a database and an ontology

In general, concept instances in the ontology usually correspond to the records of a database view or table. Furthermore, if the database table contains a primary key, then, the instance identifier (its URI) can be usually obtained by applying a transformation function to the value of that primary key in the corresponding record. The properties of a concept usually map database columns under certain conditions and after certain transformations.

Barrasa-Rodriguez et al. [8] also present different mapping situations arising from database-to-ontology mapping scenarios which may occur when a database is mapped to an existing ontology:

- A database table directly maps a concept in the ontology. Every record of the table correspond to an instance of an ontology concept.
- A database table maps more than one concept in the ontology, but only one instance per concept. Each record of the table correspond to an instance of each concept.
- A database table maps more than one concept in the ontology, but multiple instances per concept. Each record of the table correspond to one or more instances of each concept.
- A set of database tables map a concept in the ontology when they are joined. Every join record of the joined tables correspond to an instance of an ontology concept (join/union).
- A subset of the columns of a database table maps a concept in the ontology (projection).
- A subset of the rows of a database table map a concept in the ontology (selection).

### 5.1.4 Data Migration

In ontology-based information integration, the intended flow of data is from the database to the ontology. That is the reason why mappings define how to create instances in the ontology in terms of the data stored in the database, which is the information source. However, the migration of database instances into ontological instances (individuals), also called ontology population, may be done in two ways/modes [9] (Figure 5.2):

- **Massive dump (batch)** – In this mode, an independent semantic repository is created as instance of the ontology being mapped. All the database instances are then migrated to this repository. Users can query directly this ontological repository. The advantage of this mode is its high performance in query answering since it does not require any interference with local processing at source. However, it duplicates the data instances, so this may cause data inconsistencies when the data source changes. If the source database changes rapidly, the ontological repository becomes out-of-date, causing inadequate query answers.
- **Query driven (on demand)** – In this mode, ontology instances are only created to answer queries, that is, only the database instances needed to answer a given query are transformed to ontology instances. This on-demand mode always guarantees fresh and up-to-date information and it is adequate for changing data sources. However, in this mode query answering is slower than in massive dump mode, because it requires local processing at source. In general, query driven population requires methods and tools to translate user queries (over the ontology) to the local source querying language (SQL).

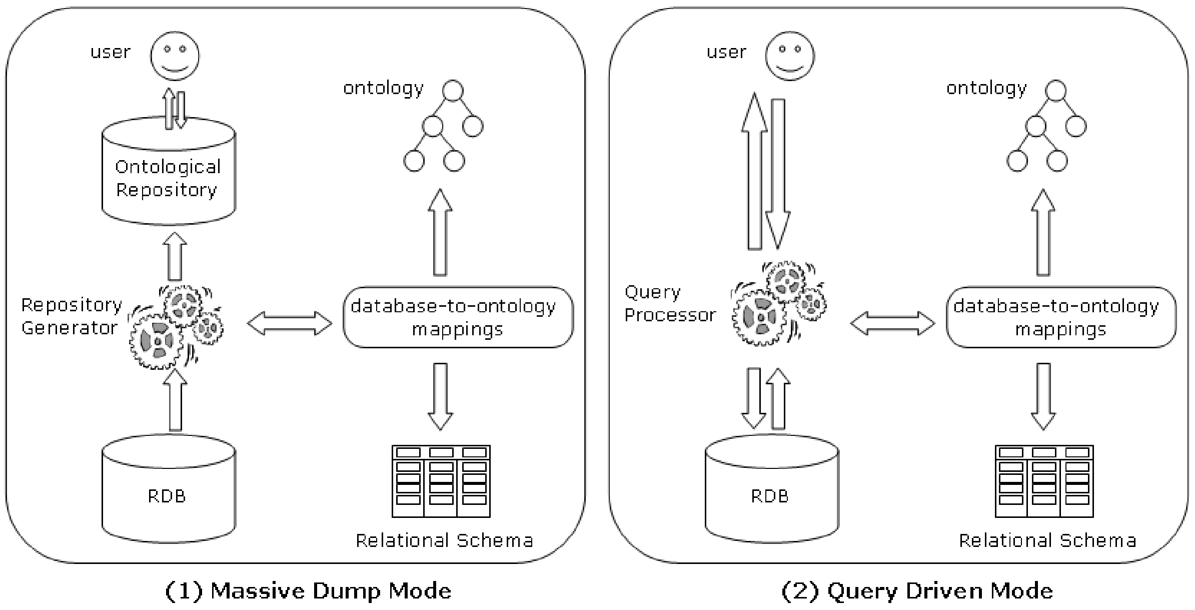


FIGURE 5.2: Two modes for data migration

### 5.1.5 Automation

As we have seen, there are two main categories of database-to-ontology mapping approaches: approaches that create an ontology from a database, and approaches that map a database to an existing ontology.

In the first category, some approaches allow automatic ontology creation from a database, such as the mapping generator provided by D2RQ platform (Section 5.3.1). Other approaches require a user interaction to complete the ontology creation process. For example, in Stojanovic et al. approach (Section 5.3.3), when several mapping rules could be applied, the user has to decide which one to apply.

In the second category, where there is a given ontology to which a database needs to be mapped, some approaches allow a manual definition of mappings, such as D2RQ (Section 5.3.1), R2O (Section 5.3.2), and VisAVis (Section 5.3.4). Other approaches such as MAPONTO (Section 5.3.5) follow a semi-automatic mapping procedure, where some mapping correspondences can be detected based on some user's input.

### 5.1.6 Querying

When mappings are specified between a database and ontology, queries can be formulated based on the ontology and executed in the underlying database. Most of the effort in this area has been devoted to 1) the reformulation of ontology-based queries in terms of the available information sources, and 2) the integration of results obtained from these queries. However, these studies have left open the problem of implementing wrappers that transform this data to the ontology vocabulary. Usually, these wrappers have to be implemented manually.

In the following section, we start our literature review by surveying existing database-to-ontology mapping specification languages.

## 5.2 Database-to-Ontology Mapping Specifications

In the literature, there are several mapping languages for specifying mappings between relational databases and ontologies, including D2RQ, R2O, and Relational.OWL. In the following subsections we present these languages with their main structures and primitives. We attempt to give a relatively detailed picture about these languages because we will build upon them our mapping specification language DOML as we will see in Section 6.3.

### 5.2.1 D2RQ

The D2RQ[2] Mapping Language [25] is an RDF-based declarative language for describing the relation between a relational data model and an OWL/RDFS ontology. The D2RQ Platform (see Section 5.3.1) uses these mappings to enable applications to access an RDF-view on a non-RDF database.

In D2RQ, basic concept mappings are defined using class maps (`d2rq:ClassMap`) that assign ontology concepts to database sets. A class map represents a class or a group of similar classes of an OWL ontology or RDFS schema, and specifies how instances of the class (or group of classes) are identified. Instances can be identified using URI references or blank nodes. URI references can be created with `d2rq:uriColumn` and `d2rq:uriPattern` primitives. Blank nodes are created using `d2rq:bNodeIdColumns` primitive.

A class map is connected to a `d2rq:Database` which represents the database where instance data is stored. A `d2rq:Database` defines a JDBC or ODBC connection to a local relational database and specifies the type of the database columns used by D2RQ. A D2RQ map can contain several `d2rq:Databases` for accessing different local databases.

[2]http://www4.wiwiss.fu-berlin.de/bizer/d2rq/spec/index.htm

A class map has also a set of property bridges (`d2rq:PropertyBridge`), which relate database table columns to RDF properties. Property bridges specify how the properties of an instance are created. They are used to construct URIs, blank nodes and literals from database values and attach them to properties of instances.

There are two types of property bridges: 1) datatype property bridges, which are used to construct literals from database values, and 2) object property bridges, which are used to construct URIs from database values and to refer to instances of other class maps. However, both cases are handled by the `d2rq:PropertyBridge` class. The distinction is made by using an appropriate property on the bridge declaration: `d2rq:column` and `d2rq:pattern` for literals, `d2rq:uriColumn`, `d2rq:uriPattern` and `d2rq:bNodeIdColumns` for resources.

Object property bridges should have a `d2rq:refersToClassMap` primitive that indicates the related concept bridge, and a `d2rq:join` primitive that indicates how the related tables have to be joined together.

A `d2rq:TranslationTable` is an additional layer between the database and the RDF world. It translates back and forth between values taken from the database and RDF URIs or literals. A translation table can be attached to a class map or a property bridge using the `d2rq:translateWith` property.

### 5.2.2 R2O

$R_2O$ (Relational to Ontology) [8] is an extensible, declarative XML-based language for describing mappings between relational database schemas and ontologies implemented in RDF or OWL. $R_2O$ is fully declarative and extensible while expressive enough to describe the semantics of the mappings. In general, like D2RQ, it allows the definition of explicit correspondences between components of two models, but $R_2O$ is more expressive as it provides an extendable set of condition and transformation primitives. $R_2O$ is an RDBMS independent high level language that works with any database implementing the SQL standard. An $R_2O$ mapping defines how to create instances in the ontology in terms of the data stored in the database.

A mapping description in $R_2O$ consist of the following components:

1. `ontology` – an ontology URI for which instances will be generated.
2. `dbschema-desc` – a database definition. Only the main structural elements of a relational database are taken into account: tables, columns, keys, and foreign keys.
3. `conceptmap-def` – one or more concept mapping definitions.

#### Database schema description

A database schema description (`dbschema-desc`) provides a description of the main structural elements in a database schema to be mapped with an ontology. A `dbschema-desc` consists of the following components:

1. `name` – the name of a database schema (a string).
2. `has-table` – one or more table descriptions, each of them representing a table in the database.

A table description provides a description of a database table, and it consists of the following components:

1. `name` – the name of the table (a string).
2. `tableType` – the type of table: System table, User table, or View.
3. `hasColumn` – one or more column descriptions (`column-desc`), each of them describes a column in the table.

Columns can be either key columns `keycol-desc`, foreign key columns `forkeycol-desc` or non key columns (the rest of them) `nonkeycol-desc`. Any of them consist of the following components:

1. `name` – the name of the column (a string).
2. `columnType` – the column data type.
3. `refers-to` – establishes the key column referred by a foreign key. Obviously this element will only occur in `forkeycol-desc` column descriptions.

### Definition of concept mappings

A concept mapping definition (`conceptmap-def`) associates an ontology class name belonging to the target ontology (therefore it must be defined in the target ontology) with a description of how to obtain it from the database. A `conceptmap-def` consists of the following components:

1. `name` – the identifier of a concept in the target ontology (URI of the class).
2. `identified-by` – one or more columns (previously described with a `column-desc` element) that identify the concept uniquely in the database.
3. `uri-as` – a pattern expressed in terms of transformation elements describing how URIs for the new instances extracted from the database will be generated. URIs will normally be obtained from the key fields in a row.
4. `described-by` – zero or more `propertymap-def`, each of them representing how the properties (attributes and relations) of the concept being defined are obtained.
5. `applies-if` – contains a condition expression `cond-expr` describing under which conditions the mapping is applicable.
6. `joins-via` – contains a `join-list`, describing how the different tables implied in the description of a concept mapping are joined.

### Attribute and relation mappings

A property mapping description (attributes and relations) associates an attribute or a relation name belonging to a target ontology concept with a description of how to obtain it from the database. Depending on the type of property, these kinds of mappings can either be described with an `attributemap-def` element, a `relfromatt-def` element or a `relationmap-def` element. The former describes attribute mappings and the two laters describe relations. An `attributemap-def` contains the following components:

1. `name` – the identifier of a property (attribute or relation) in the target ontology (its URI).
2. `use-dbcol` – zero or more database columns (previously described with a `column-desc` element) that participates in obtaining the property being defined. Zero columns are accepted if there is at least one `aftertransform` element.
3. `to-concept` – the name of the concept mapping element (previously defined as such, and consequently described in terms of some database tables) to which this property will be a link. This information should be enough to find out the link between tables implied in both the definitions of the source and target concepts of the relation.

### 5.2.3 Relational.OWL

Relational.OWL [53] is an OWL ontology[3] which describes the schema of a relational database in an abstract way. Based on this ontology, the schema of (virtually) any relational database can be described. In turn, this ontological representation of the schema can be used to represent the data stored in that specific database. That is, the data items are represented as instances of this data source specific ontology.

Relational.OWL, using the techniques provided by OWL, defines classes like `Table` or `Column` and specifies possible relationships among these classes. In particular, Relational.OWL includes a representation for tables, columns, datatypes (possibly with length restrictions), primary keys, foreign keys, and the relations among each other.

A summary of all the classes represented in the Relational.OWL ontology is provided in Table 5.1. Table 5.2 contains a list of the relationships, which interconnect these classes.

Table 5.1: Classes defined in the Relational.OWL ontology

| `rdf:ID` | `rdfs:subClassOf` | `rdfs:comment` |
|---|---|---|
| `dbs:Database` | `rdf:Bag` | The class of databases. |
| `dbs:Table` | `rdf:Seq` | The class of database tables. |
| `dbs:Column` | `rdfs:Resource` | The class of database columns. |
| `dbs:PrimaryKey` | `rdf:Bag` | The Primary Key of a Table. |

Table 5.2: Properties defined in the Relational.OWL ontology

| `rdf:ID` | `rdfs:domain` | `rdfs:range` | `rdfs:comment` |
|---|---|---|---|
| `dbs:has` | `owl:Thing` | `owl:Thing` | A Thing can have other Things inside. |
| `dbs:hasTable` | `dbs:Database` | `dbs:Table` | A Database has a set of Tables. |
| `dbs:hasColumn` | `dbs:Table` | `dbs:Column` | A Table has a set of Columns. |
| `dbs:isIdentifiedBy` | `dbs:Table` | `dbs:PrimaryKey` | A Table is identified by a Primary Key. |
| `dbs:references` | `dbs:Column` | `dbs:Column` | Foreign Key relationship between Columns. |
| `dbs:length` | `dbs:Column` | `xsd:nonNegativeInteger` | Maximal length of an entry in that Column. |
| `dbs:scale` | `dbs:Column` | `xsd:nonNegativeInteger` | The scale an entry of the Column may have. |

[3] http://www.dbs.cs.uni-duesseldorf.de/RDF/relational.owl

A schema represented with Relational.OWL is used itself as a novel ontology for creating a representation format for the corresponding data items. Thus, an automatic transformation mechanism is created, from data stored in relational databases into OWL representation which can be processed by virtually any Semantic Web application.

***Summary***

In this section we have reviewed some existing database-to-ontology mapping specification languages. A detailed discussion on the presented languages is given in Section 5.4.

In the next section, we will continue our literature review by investigating some of existing database-to-ontology mapping approaches. We draw the attention of the reader to that some of these approaches use the mapping languages that we have presented.

## 5.3 Database-to-Ontology Mapping Approaches

In this section, we give an overview on existing approaches for database-to-ontology mapping. These approaches include: D2RQ Platform, R2O and ODEMapster, Stojanovice et al., VisAVis, and MAPONTO.

### 5.3.1 D2RQ Platform

D2RQ [25] is a declarative language to describe mappings between relational database schemata and OWL/RDFS ontologies. The D2RQ Platform uses these mapping to enables applications to access a RDF-view on a non-RDF database through the Jena [40] and Sesame [36] Semantic Web development toolkits, as well as over the Web via the SPARQL Protocol [44] and as Linked Data [23][24]. D2RQ builds on the above concept of D2R MAP, thus retaining the `ClassMap` element, but with a slightly different syntax.

With the use of D2RQ, an application can query a non-RDF database using Semantic Web query languages like RDQL [141] or SPARQL [138], or using find (subject, predicate, object) statements. D2RQ rewrites queries and Jena API calls into database-specific SQL queries. The result sets of these SQL queries are transformed into RDF triples which are passed up to the higher layers of the Jena framework. D2RQ offers a flexible access mechanism to the content of huge, non-RDF databases without having to replicate the database into RDF.

The D2RQ Platform consists of:

- **D2RQ Mapping Language** – a declarative mapping language for describing the relation between an ontology and an relational data model (see Section 5.2.1).
- **D2RQ Engine** – a plugin for the Jena and Sesame Semantic Web toolkits, which uses the mappings to rewrite Jena and Sesame API calls to SQL queries against the database and passes query results up to the higher layers of the frameworks.
- **D2R Server** – an HTTP server that can be used to provide a Linked Data view, a HTML view for debugging and a SPARQL Protocol endpoint over the database.

D2R Server[4] is a tool for publishing relational databases on the Semantic Web. It enables RDF and HTML browsers to navigate the content of the database, and allows applications to query the database using the SPARQL query language. D2R Server builds on the D2RQ Engine.

D2R Server provides SPARQL access to relational databases. It takes SPARQL queries from the Web and rewrites them via a D2RQ mapping into SQL queries against a relational database. D2R Server can be used to integrate existing databases into RDF systems, and to add SPARQL interfaces to database-backed software products.

### 5.3.2 R2O and ODEMapster

As mentioned in Section 5.2.2, R2O [8] is an extensible, declarative, XML-based language to describe mappings between relational database schemas and ontologies implemented in RDF or OWL. An R2O mapping defines how to create instances in the ontology in terms of the data stored in the database.

The approach suggested by the authors consists of a manual creation of a mapping description document using R2O with all the correspondences between the components of the database schema and those of the ontology. Such mappings are then processed automatically by a mapping processor called ODEMapster.

ODEMapster is a generic query engine that automatically populates the ontology with instances extracted from the database content. The mapping definition and execution can be done in two modes: 1) query driven, i.e. parsing a specific query and translating its result, or 2) massive dump, i.e. creating a semantic RDF repository and translating the full database to it.

The mapping execution process starts by parsing the query and the R2O mapping document. Then, the query is translated into data source's SQL. The generated SQL query executed by the DBMS. Finally, the retrieved results are post-processed and ontology instances are generated.

A transformation function $\chi$ is used to transform R2O expressions into SQL sentences. This transformation function $\chi$ takes an expression in R2O (containing concept mappings, attribute and relation mappings with their transformations and condition expressions, etc.) and transforms it to an SQL query for a specific DBMS. Any mapping definition in R2O has an equivalent database view definition.

### 5.3.3 Stojanovic et al.

Stojanovic et al. [149][148] propose an integrated and semi-automatic approach for generating shared-understandable metadata of data-intensive Web applications. This approach is based on mapping a given relational schema into an ontology structure using a reverse engineering process.

The input of the migration architecture is a relational model that is derived from the SQL DDL. The database schema is mapped into an ontology (expressed in F-Logic) using a mapping process which applies a set of mapping rules.

Database instances that are mapped into the knowledge base, based on the domain ontology. The actual ontology is computed once (under the supervision and revision of the designer) and

[4]http://www4.wiwiss.fu-berlin.de/bizer/d2r-server/

must be recomputed only if the database schema is changed. Knowledge base instances are computed on demand. Servlets are used to create the RDF output files from the ontology and the database data. Two files are produced: one file containing the ontology and another file containing instances that refers to the schema file using XML namespace mechanisms. To create the files F-Logic is mapped into RDF.

The mapping process enhances the semantics of the database by providing additional ontological entities. The proposed method for ontology creation from databases consists of five steps:

1. Capture information from a relational schema through reverse engineering (consider relations, attributes, primary and foreign keys).
2. Analyse the obtained information to construct ontological entities, using a set of mapping rules.
3. Form an ontology ("schema translation") by applying the mentioned rules.
4. Evaluate, validate and refine the ontology.
5. Form a knowledge base ("data migration"), involving the creation of ontological instances based on the tuples of the relational database.

The implemented system provides assistance in all phases. Actually, the reverse engineering process cannot be completely automated as some situations can arise where several rules could be applied. User interaction is then necessary when this kind of ambiguities occur and domain semantics cannot be inferred.

### 5.3.4 Vis-A-Vis

VisAVis [106] is a manual schema-to-ontology converter implemented as a plug-in to the popular ontology editor Protégé [125]. Vis-A-Vis allows to map relational databases to existing ontologies. Mapping is done by selecting from the database a dataset corresponding to an ontology class. A new property is added to the class which consists of an SQL query which will be executed and return the desired dataset. This tool also performs a set of consistency checks to insure the validation of mappings.

The mapping process consists of four steps :

1. Capture data from the database. The data can be any combination of datasets (records belonging to various tables).
2. Select a class of ontology.
3. Consistency checks and evaluation, validation and refinement of the mapping.
4. Modifications of the resulting ontology and the additional information are added to it.

These four steps can be repeated as many times as required.

The final result is the initial ontology enhanced to include references to datasets. These references will be under the form of class properties in the ontology, all assigned as value a string containing the actual SQL query that returns the dataset. Each query posed to the ontology will check the existence of the mapping property. In the case that it does not exist, the results are no affected. Otherwise, the results will be database entries instead of class individuals.

### 5.3.5 MAPONTO

MAPONTO[5] [3] is a semi-automatic tool that assists users to discover plausible semantic relationships between a database schema and an ontology, expressing them as logical formulas. MAPONTO is implemented as a tab plugin for the popular ontology editor Protégé [125].

Inspired by the Clio project [86], MAPONTO is built to work in an interactive and semi-automatic manner. Starting with a set of simple attribute-to-attribute correspondences, the tool analyzes semantics in the two input data models (e.g., a schema and an ontology, or two schemas), and then generates a set of logical formulas representing a semantic relationship between the two data models. The final list of logical formulas are ordered by the tool with the most "reasonable" mappings between the two models on top. Finally, the user could choose the expected mapping from the list.

This tool expects the user to provide simple correspondences between atomic elements used in the database schema (e.g., column names in tables) and those in the ontology (e.g., attribute/-datatype property names on concepts).

Given the set of correspondences, MAPONTO is expected to reason about the database schema and the ontology, and to generate a list of candidate rules for each individual component (e.g., a table) in the database schema.

The main idea underlying MAPONTO is to represent the ontology as a graph consisting of nodes (concepts) connected by edges (properties). Then, the tool finds the minimum spanning tree (Steiner tree[6]) connecting the concepts having datatype properties corresponding to table columns, and encodes the tree into a logical formula by joining the concepts and properties encountered in it.

## 5.4 Discussion

In the first section of this chapter, we have presented the main issues in database-to-ontology mapping. These issues include: the context within which the mapping approach is used, the nature of the target ontology, the mapping definition, the instance migration, the automation level, and the querying process. Then, we have made a literature review to investigate existing works on database-to-ontology mapping languages and approaches. In this section, we will discuss those works according to the presented issues. Firstly, we will discuss mapping specification languages, then we discuss mapping approaches.

### Mapping Specification Languages

In Section 5.2, we reviewed some database-to-ontology mapping specification languages, namely: D2RQ, R2O, and Relational.OWL. The first thing we observe is that the three languages support ontologies expressed in RDF or OWL. However, these languages are themselves expressed in different formulations: R2O is a based on XML, D2RQ is based on RDF, while Relational.OWL

[5]http://www.cs.toronto.edu/semanticweb/maponto/

[6]A Steiner tree for a set M of nodes in graph G is a minimum spanning tree of M that may contain nodes of G which are not in M.

is based on OWL. This point reveals the level of expressiveness of each language according to the used formulation.

We note also that Relational.OWL and R2O provide means to describe relational schemas. But D2RQ include only means to describe how to connect to the database (without description of the relational schema).

Another interesting point should also be noticed, that Relational.OWL essentially differs from the other languages. In fact, the aim of D2RQ and R2O languages is to allow the specification of mappings between a database and an existing ontology. Both languages provide means to specify concept mappings and property mappings. However, in contrast to D2RQ and R2O, Relational.OWL is attended to reformulate a relational schema using OWL language, and it does not serve to specify mappings between a relational schema and an existing ontology.

Our last observation is about transformations and conditional mappings. We can notice that R2O provides primitives for specifying transformations and conditional mappings on the fly. Whereas, D2RQ does not support transformations, it allows to specify conditional mappings (using `d2rq:condition`), but still limited to SQL syntax. Relational.OWL does not support neither conditions nor transformations.

In summary, we find that each language has its advantages and shortcomings. We think that the main advantage of Relational.OWL over other languages is its capability to describe relational schemas. The advantage of D2RQ is its way to form concept and property mapping bridges. Finally, The main advantage of R2O is its set of primitives to describe transformations and conditional mappings.

We think that a combination of these languages will make the best of them and profit from the advantages of each one. In Section 6.3, we propose a database-to-ontology mapping language (called DOML) which is mainly a combination of these three languages.

### Mapping Approaches

In Section 5.3, we have made a literature review about existing database-to-ontology mapping approaches. Reflecting the presented approaches on the issues presented in Section 5.1, we obtain Table 5.3.

From this survey, the first point that we observe concerns the nature of the target ontology. We find that some approaches allow only to map a database to an existing ontology, such as: R2O, VisAVis, and MAPONTO. However, some approaches provide both possibilities: ontology creation from a database, and mapping a database to an existing ontology. For example, D2RQ Platform allow users to manually provide mappings as a document written in D2RQ language, and it offers, as well, a mapping generator that can automatically generate such a mapping document from a database. In Stojanovic et al approach, the main goal is to create an ontology from a database, however, it is also possible to manually map a database to an existing ontology using KAON-Reverse tool[7].

We have seen in Section 5.1.3 that the mapping definition varies according to the nature of the target ontology, that is, whether it is created from the database or it already exists:

[7] http://kaon.semanticweb.org/alphaworld/reverse/

- In approaches that create an ontology from a database such as Relational.OWL, DataGenie and Stojanovic et al. approach, the database model and the structure of the generated ontology are very similar. However, the creation of ontology structure may be done in two possible ways. The first way is simple, and based on straightforward transformation of every database table into an ontology concept and every column into a property. Relational.OWL follows this approach. The second way is more complex and involves the discovery of additional semantic relations between database components (like the referential constraints) and take them into account while constructing ontology concepts and relations between them. DataGenie and Stojanovic et al. are examples of this approach.

- In approaches that map a database to an existing ontology such as R2O, Vis-A-Vis, MAPONTO, and D2RQ Platform, mappings are more complex than those in the previous case. Complex mapping scenarios usually arise from low similarity between the ontology and the database model (one of them is richer, more generic or specific, better structured, etc., than the other). However, the presented approaches use different ways define mappings in order to cope with complex scenarios. The first way is followed by VisAVis approach, where a suitable SQL query is directly assigned to each ontology concept. The instance data retrieved by this query are used to instantiate the concept. The second way is followed by D2RQ and R2O approaches, where mappings are expressed as concept-to-table and property-to-column bridges. Such bridges are then translated into appropriate SQL statements. The last way is followed by MAPONTO approach, where mappings are deduced automatically basing on simple correspondences provided by the user. The technique used to deduce mappings is based on shortest path finding between concepts of the ontology.

The level of automation also varies according to whether the ontology is created from the database or it already exists. In the first case, some approaches allow automatic ontology creation from a database, such as Relational.OWL, and the mapping generator provided by D2RQ platform. Other approaches are semi-automatic and require a user interaction to complete the ontology creation process. For example, in Stojanovic et al. approach, when several mapping rules could be applied, the user has to decide which one to apply. In the second case, where there is a given ontology to which a database needs to be mapped, some approaches allow a manual definition of mappings, such as D2RQ, R2O, and Vis-A-Vis. Other approaches such as MAPONTO follow a semi-automatic mapping procedure, where some mapping correspondences can be detected based on some user's input.

We have seen also that there are two modes for data migration: massive dump and query-driven modes. We observe that Stojanovic et al. approach use the massive dump mode. Whereas, Vis-A-Vis use the query-driven mode. Some approaches such as D2RQ and R2O offer the possibility to use both modes. However, no one of approaches that use query-driven mode describe its method for translating queries over the ontology into SQL queries.

### Conclusion

From observing these approaches, we learn some features that we will adopt in our mapping tool that we use within OWSCIS system. Our mapping tool is called DB2OWL and it will be presented in Chapter 7. This tool should allow both processes: 1) the creation of an ontology from a database and 2) mapping a database to an existing ontology.

For creating an ontology from a database, we will use some mapping rules that are similar to those used in Stojanovic et al. approach. That is, our tool will analyze the database components (tables, columns, primary/foreign keys), and suitably transform them into ontology components. In addition, the process will involve the discovery of additional semantic relations between database components (like the referential constraints) and take them into account while constructing ontology concepts and relations between them.

For mapping a database to an existing ontology, we will use a manual mapping definition. In fact, we will not attempt to investigate a semi-automatic approach because we think that such approach will only yield simple and direct one-to-one mappings. However, we want more sophisticated mappings such as one-to-many or many-to-one, as well as, transformation-based and conditional mappings. Thus, we believe that such mappings cannot be deduced automatically or semi-automatically, therefore, we assume that the user should provide them. Our tool should offer a graphical interface that allow users to manually specify sophisticated mappings.

Describing such sophisticated mappings requires a rich mapping specification. In our tool, we use two possible mapping specifications. The first one is similar to that used by D2R MAP and VisAVis, that is, associating ontology components with SQL statements (this specification will be presented in Section 6.2). The second one is a more sophisticated mapping specification language that we propose basing on D2RQ, D2R and Relational.OWL languages. This language (called DOML) will be presented in the next chapter in Section 6.3.

Finally, our tool will use the query-driven mode for data migration. In fact, the reason behind that is because we use a non-materialized approach within OWSCIS system. Thus, we don't want to use an ontology repository to store instances. Consequently, we will need an declarative and explicit method to translate queries over ontologies (in SPARQL language) into SQL queries over the database. This method will be presented in Chapter 8.

## Chapter Summary

In this chapter, we have given a global background and the state of the art of the field of database-to-ontology mapping.

In Section 5.1, we have presented the main issues in database-to-ontology mapping. These issues include: the context within which the mapping approach is used, the nature of the target ontology, the mapping definition, the instance migration, the automation level, and the querying process.

In Section 5.2, we have made a literature review about some database-to-ontology mapping specification languages such as D2RQ, R2O, and Relational.OWL. In Section 5.3, we have investigated several database-to-ontology mapping approaches.

Finally, in Section 5.4, we have discussed the presented mapping languages and approaches. We have concluded that we need, within OWSCIS system, a mapping specification language and a mapping tool. Our proposed mapping language, called DOML, will be presented in Section 6.3, whereas our mapping tool, called DB2OWL, will be presented in Chapter 7.

Table 5.3: Main features of database-to-ontology mapping approaches

| **Tool** | **Target ontology** | **Methodology / Techniques** | **Components mapped** | **Data migration** | **Degree of automation** |
|---|---|---|---|---|---|
| **D2RQ** | Both existing or created | Languages for mappings description | DB tables, columns, primary / foreign keys | Both massive-dump and query-driven | Both manual and automatic |
| **R2O** | Existing | Ontology populated with instances according to a set of mappings specified by the user | DB tables, columns, foreign keys | Both massive-dump and query-driven | Manual |
| **Stojanovic et al.** | Both existing or created | Table to Class and Column to Property, some exceptions exist | DB tables, columns, integrity constraints, keys | Massive-dump | Semi-automatic |
| **VisAVis** | Existing | Add SQL query as new property to ontology concepts | DB tables, columns, foreign keys | Query-driven | Manual |
| **MAPONTO** | Existing | Shortest path finding between concepts of the ontology | DB tables and columns | None | Semi-automatic |

# Chapter 6

# Database-to-Ontology Mapping Specification

**Contents**

## Abstract

In the previous chapter, we gave a background on the topic of database-to-ontology mapping. We basically made a literature review about existing database-to-ontology mapping languages and approaches. In this chapter, we propose two database-to-ontology mapping specifications:

- Associations with SQL statements, and
- DOML language.

In section 6.1, we give a brief introduction to these specifications.

In Section 6.2, we present the first mapping specification: *associations with SQL statements*. In this kind of mappings, we distinguish:

1. Concept Associations (Section 6.2.1),
2. Object Property Associations (Section 6.2.2), and
3. Datatype property Associations (Section 6.2.3)

In Section 6.3, we present the second mapping specification: *DOML language*. We firstly motivate the need to this language (Section 6.3.1), and give a general description of it (Section 6.3.2). Then we illustrate how to use DOML to:

1. describe relational schemas (Section 6.3.3),
2. specify concept and property mapping bridges (Section 6.3.4), and
3. specify transformation and condition rules (Section 6.3.5).

## 6.1 Introduction

We saw in Chapter 3, that OWSCIS system includes a tool call DB2OWL for wrapping relational databases to local ontologies within data providers. That is, this tool allows three main processes: 1) creation of an ontology from a database and 2) mapping a database to an existing ontology, and 3) translation of SPARQL queries into SQL queries. DB2OWL tool is described in details in the next chapter. However, we firstly present, in this chapter, the two database-to-ontology mapping specification used in DB2OWL tool, namely, 1) associations with SQL statements, and 2) DOML language.

### 1) Associations with SQL Statements

We have seen in Section 5.2, that in the literature there are several mapping languages to specify database-to-ontology mappings. These languages include: D2RQ, R2O and Relational.OWL.

Among these languages, we chose R2O (Section 5.2.2) as a candidate mapping language to use within DB2OWL tool[1]. However, R2O mappings have to be translated into a set of SQL statements for each ontology concept or property.

[1]In the very first version of DB2OWL (April, 2007), mappings were actually specified in R2O [51].

Therefore, in order to simplify this process, we decided to directly specify the mappings as associations with SQL statements, without using R2O language.

The approach of "*Associations with SQL Statements*" is a simple approach in which ontology terms (concepts and properties) are simply mapped to SQL statements over the database. This mapping specification is presented in details in Section 6.2.

### 2) DOML Language

Recently, we investigated another mapping specification language called DOML[2]. This language is adapted from D2RQ, R2O and Relational.OWL and is based on RDF. It allows the specification of sophisticated database-to-ontology mappings that cannot be expressed using associations with SQL statements.

DOML mapping language is presented in details in Section 6.3.

## 6.2 Associations with SQL Statements

In this kind of mapping specifications, all mapping are expressed as associations (correspondences) between ontology components (concepts and properties) and SQL statements. This kind of mappings is very useful for translating SPARQL queries over the ontology into SQL queries over the database as we will see in Chapter 8.

### 6.2.1 Concept Associations

A concept association (CA) associates an ontology concept with an SQL statement that retrieve some identifier that uniquely identify the instances of that concept. Such identifier is usually a primary key of some database table (corresponding to the concept).

However, the SQL statement associated with a concept must have exactly one item in the SELECT clause. This item represents the identifier of the concept and always given an alias "`DOM`".

**Example** – Let us consider a concept `ex:Paper` that corresponds to a table `Paper` whose primary key is the column `paperId`. The mapping between this concept and table is expressed using a concept association that associates the concept `ex:Paper` with the following SQL statement:

```
SELECT Paper.paperId AS DOM FROM Paper
```

This concept association can be described in XML format as follows:

```
<conceptAssociation>
  <uri>uribase#Paper</uri>
  <sql>SELECT Paper.paperId AS DOM FROM Paper</sql>
  <dom>Paper.paperId</dom>
</conceptAssociation>
```

[2]DOML stands for: **D**atabase to **O**ntology **M**apping **L**anguage.

**Property Associations**

A property association (PA) associates an ontology property with an SQL statement that has exactly two items in the SELECT clause. These two items represent the domain and the range of the corresponding property. That is, the SQL statement typically retrieves all the pairs ⟨domain, range⟩ of the property from the database.

The first item is the identifier of the domain concept of the property and always given an alias "`DOM`". The nature of the second item varies according to the kind of property being mapped, i.e., whether it is an object property or a datatype property. However, it is always given an alias "`RNG`".

### 6.2.2 Object Property Associations

For object properties, the second item of the associated SQL statement is the identifier of the range concept of the object property. This means that an SQL statement associated with an object property will retrieve the (identifiers of) pairs of instances related by this object property.

**Example** – Let us consider two concepts `ex:Paper` and `ex:Publisher`, and an object property `ex:publishedBy` that indicates the publisher of a paper (the domain of this object property is `ex:Paper` and its range is `ex:Publisher`).

The concept `ex:Paper` corresponds to a table `Paper` whose primary key is the column paperId. The concept `ex:Publisher` corresponds to a table `Publisher` whose primary key is the column `publisherId`. The table `Paper` has a foreign key `publisher` that references `publisherId`.

The mapping of the object property `ex:publishedBy` is expressed using an object property association that associates this property with the following SQL statement:

```
SELECT Paper.paperId AS DOM, Publisher.publisherId AS RNG
FROM Paper, Publisher
WHERE Paper.publisher = Publisher.publisherId
```

This object property association can be described in XML format as follows:

```
<objectPropertyAssociation>
  <uri>uribase#publishedBy</uri>
  <sql>SELECT Paper.paperId AS DOM, Publisher.publisherId AS RNG
       FROM Paper, Publisher
       WHERE Paper.publisher = Publisher.publisherId</sql>
  <dom>Paper.paperId</dom>
  <rng>Publisher.publisherId</rng>
</objectPropertyAssociation>
```

### 6.2.3 Datatype Property Associations

For datatype properties, the second item of the associated SQL statement indicates the values of this datatype property for the instances identified by the first item (the identifier of the domain concept). These values usually come from the database column corresponding to the datatype property. This means that an SQL statement associated with a datatype property will retrieve the pairs ⟨instance, value⟩ related by this datatype property.

**Example** – Let us consider a concept `ex:Paper` that corresponds to a table `Paper` whose primary key is the column `paperId`. This concept has a datatype property `ex:title` that corresponds to the column `Paper.title`. The mapping between this datatype property and column is expressed using a datatype property association that associates the concept `ex:title` with the following SQL statement:

```
SELECT Paper.paperId AS DOM, Paper.title AS RNG
FROM Paper
```

This datatype property association can be described in XML format as follows:

```
<datatypePropertyAssociation>
  <uri>uribase#title</uri>
  <sql>SELECT Paper.paperId AS DOM, Paper.title AS RNG
       FROM Paper</sql>
  <dom>Paper.paperId</dom>
  <rng>Paper.title</rng>
</datatypePropertyAssociation>
```

#### Complete Example

Appendix C presents a complete example on *associations with SQL statements* mappings.

### 6.2.4 Discussion

The main advantage of using "*Associations with SQL statements*" as mapping specifications is to facilitate the SPARQL-to-SQL translation (as we will see in Chapter 8). However, this kind of mapping specifications has some limitations.

First of all, we cannot specify transformations or conditional mappings explicitly. Although, it is possible to specify some basic transformations and conditions implicitly within the SQL statements, but this is still unintuitive to users and limited to SQL syntax.

The second limitation is the dependence on primary keys to indicate concept identifiers. Thus, in the cases where a database table have a compound primary key (more than one column), or does not define a primary key, the specification of associated SQL statements becomes difficult.

In order to overcome these limitations, we had to find an alternative mapping specifications for our tool DB2OWL. The alternative is our DOML language proposed in the next section. The advantage of this language is that it allows the specification of sophisticated mappings that include transformations and/or conditions, and it does not depend on primary keys.

## 6.3 DOML Language

### 6.3.1 Motivation

We have seen in the previous chapter, that there are in the literature several database-to-ontology mapping specification languages, such as D2RQ, R2O and Relational.OWL. In Section 5.4, we discussed these languages and argued that each language has its advantages over other languages.

We think that a combination of these languages will make the best of them and profit from the advantages of each one. In this chapter, we propose a database-to-ontology mapping language (called DOML) which is mainly a combination of these three languages. The investigation of DOML is mainly motivated by the need to integrate the advantages of each of these languages.

For instance, the RDF-based structure of D2RQ, the primitives provided by R2O for describing conditions and transformations, and the primitives provided by Relational.OWL for describing relational schemas.

DOML is a general-purpose language to specify database-to-ontology mappings. Thus it is be exploited in contexts that needs such mappings, like ontology-based information integration, and the Semantic Web. However, in this dissertation, we will limit the exploitation of DOML to the context of ontology-based information integration since it is the main context of the dissertation. Consequently, we will not mention the role of DOML in the Semantic Web context.

As we said previously, a tool called DB2OWL (see Chapter 7) is used within OWSCIS platform to wrap a local database (at the site level) to local ontology. DOML language is attended to be used by this tool to specify sophisticated mappings between the components of the database and the ontology. We said also that, within OWSCIS we will use GAV approach at the site level to relate local ontologies to their underlying local information sources. This means that the local ontology is expressed in terms of local information sources. Consequently, the mappings in DOML language will be from the ontology to the database. That is, each ontology component is expressed in terms of (relational) database components.

In the next section, we give a general introduction to DOML language, which will be detailed in later sections.

### 6.3.2 DOML Description

DOML is an extensible, declarative RDF-based language for describing mappings between relational databases and OWL ontologies. DOML is a combination of D2RQ, R2O and Relational.OWL mapping languages. DOML derives from D2RQ its structure and basic primitives, while it derives from R2O its set of condition and transformation primitives. It derives from Relational.OWL its primitives for describing relational schemas.

DOML allows the definition of explicit correspondences between components of a relational database and an OWL ontology. These correspondences are called Mapping Bridges. A mapping document written in DOML is basically composed of a set of mapping bridges. A mapping bridge relates an ontology entity with one or more entities of the relational database.

In order to enable DOML language to specify sophisticated mappings, we consider several dimensions of a mapping bridge (inspired from [116]). In DOML, a mapping bridge is characterized by several dimensions:

- **Entity dimension** – This dimension indicates the type of ontology entity participating to the mapping bridge. We distinguish three types of bridges: 1) concept bridges (CBs), 2) datatype property bridges (DPBs), and 3) object property bridges (OPBs).
- **Transformation dimension** – A mapping bridge can relate an ontology entity to a database entity either directly or via a transformation function.
- **Cardinality dimension** – A mapping bridge can relate an ontology entity to either one database entity (one to one) or to multiple database entities (one to many).
- **Constraint dimension** – This dimension indicates any conditions that constraint the usage of the mapping bridge. A conditional mapping bridge applies only when its associated condition holds.

Like R2O and Relational.OWL, DOML language permits to explicitly describe a relational schema within the mapping document. Our purpose is to give the relational constructs the same form (RDF resources) of the mapping bridges, thus enabling them to interrelate uniformly using RDF triples.

The principle behind DOML language is to design an OWL ontology that contains all main primitives needed for describing database-to-ontology mappings. Thus, a mapping document will be an instance of this ontology, i.e., an RDF graph conforming to this ontology. In fact, DOML ontology contains primitives for describing 1) the components of a relational schema, 2) mapping bridges for concepts, object properties, and datatype properties, and 3) transformation and conditions.

In the reminder of this chapter, we present this DOML language in details along with its primitives.

**Running Example**

We will use the following example throughout this chapter in order to illustrate how DOML is used to specify database-to-ontology mappings. In this example, we have a simple database schema of two tables *Person* and *Department*, and a simple ontology of four concepts: `Person`, `Student`, `Employee`, and `Department`.

In the database, the table *Person* contains information about people of a university students and employees: the first and last name, the status (`"student"` or `"employee"`), salary (for employees only), year (for students only), email, and department in which the student studies, or the employee works.

However, in the ontology, there is an explicit distinction between students (concept `Student`) and employees (concept `Employee`). In addition, in the ontology, the name of a person is modeled as one datatype property `name`, whereas, in the database, the person name is modeled as two columns indicating the first and the last name, respectively.

We will see how to describe, using DOML, a relational database schema, concept bridges and property bridges, conditions and transformations

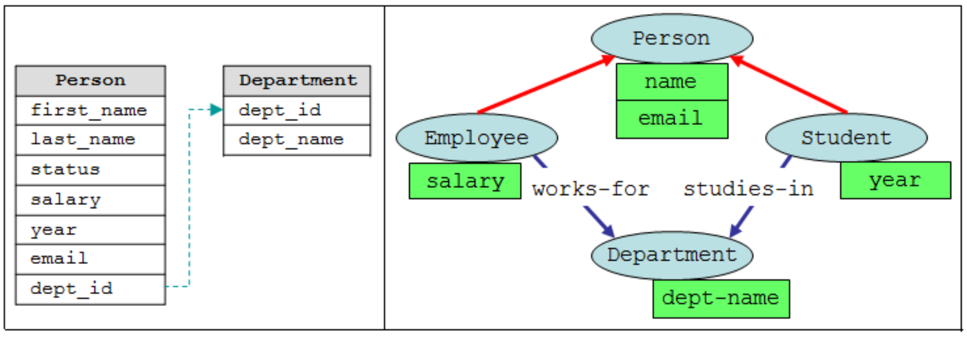


FIGURE 6.1: DOML language, running example

## 6.3.3 Describing Relational Schemas in DOML

DOML uses four primitives (borrowed from Relational.OWL language) to represent the main constructs of a relational database:

- The primitive `doml:Database` is used to define a database schema
- The primitive `doml:Table` is used for defining tables,
- The primitive `doml:Column` is used for defining columns.
- The primitive `doml:PrimaryKey` is used for defining primary keys.

The primitive `doml:Database` is used to represent a relational schema. A `doml:Database` has the following features:

1. a name (specified using `doml:name`), and
2. several tables (represented using `doml:Table`) specified using the property `doml:hasTable`.

To represent tables, the class `doml:Table` is used. A `doml:Table` has:

1. a name (specified using `doml:name`),
2. several columns (represented using `doml:Column`) specified using the property `doml:hasColumn`, and
3. a primary key (represented using `doml:PrimaryKey`) specified using the property `doml:isIdentifiedBy`.

The primitive `doml:Column` is used to represent a column. For a `doml:Column` we can specify:

1. The name (specified using `doml:name`).
2. The datatype (represented as XML schema datatype) specified using `doml:columnType`.
3. The table to which the column belongs (represented using `doml:Table`) specified using the property `doml:belongsToTable`.
4. When the column is a foreign key, we can specify the referenced column (represented as `doml:Column`) using the property `doml:references`.

The primitive `doml:PrimaryKey` is used to represent a primary key. A `doml:PrimaryKey` can have one or more columns (`doml:Column`) specified using `doml:hasColumn`.

Appendix D shows the DOML mapping document of our running example. The representation of the database schema is depicted in lines 8-76 of this document.

### 6.3.4 Describing Mapping Bridges in DOML

DOML uses different primitives to represent mapping bridges:

- The primitive `doml:ConceptBridge` is used to define mappings for concepts.
- The primitive `doml:DatatypePropertyBridge` is used to define mapping bridges for datatype properties.
- The primitive `doml:ObjectPropertyBridge` is used to define mapping bridges for object properties.

DOML uses a group of properties that defines possible relationships among mapping bridges and associated conditions and/or transformations.

#### 6.3.4.1 Concept Bridges

Concept Bridges (CBs) are used to relate ontology concepts with database tables. Each concept bridge has a unique identifier. In DOML specification, the primitive `doml:ConceptBridge` is used to represent mapping bridges for ontology concepts. This primitive has three properties (the first and the second are mandatory whereas the third is optional):

- `doml:class` that indicates the mapped ontology concept (`owl:Class`), and
- `doml:toTable` that indicates the corresponding database table (an instance of `doml:Table`).
- `doml:when` indicates an optional condition that can be associated to the concept bridge. This condition is an instance of `doml:Condition` (see Section 6.3.5.1). A conditional concept bridge means that this concept bridge is usable only when the associated condition holds.

In RDF representation of DOML, a concept bridge is encoded as follows:

```
[bridge-ID]     a                 doml:ConceptBridge;
                doml:class        [class-URI];
                doml:toTable      [table-resource];
                doml:when         [condition] .
```

where:

- `[bridge-ID]` is a resource representing the identifier of the concept bridge (an instance of `doml:ConceptBridge`).
- `[class-URI]` is the URI of the mapped class.
- `[table-resource]` is a resource representing the mapped database table (an instance of `doml:Table`), this resource should be already defined in the part of database schema specification of the mapping document.
- `[condition]` is a resource representing an optional associated condition (an instance of `doml:Condition`), this resource should be defined somewhere in the mapping document.

For simplification, a concept bridge can be noted:

$\boxed{id = CB(\mathbf{C}, \mathbf{T}, WHEN(condition))}$

where:

- *id* is the identifier of the bridge,
- the symbol *CB* is used to indicate that this bridge is a Concept Bridge
- the first argument **C** indicates an ontology term (in this case, a concept)
- the second argument **T** indicates a database term, (in this case, a table).
- the third argument indicates an optional condition.

**Example** – In our running example, we define four concept bridges:

The first concept bridge relates the concept Person to the table Person; this bridge is encoded in RDF as follows:

```
map:person-cb  a               doml:ConceptBridge ;
               doml:class      ont:Person ;
               doml:toTable    map:person-table .
```

For simplification, this bridge is noted:

$\boxed{\textit{person-cb} = \textit{CB}(\textbf{Person}, \textbf{person})}$

The second bridge relates the concept Student to the table person with a condition which is: the column *person.status* has the value: "Student". This bridge is encoded in RDF as follows:

```
map:student-cb   a              doml:ConceptBridge ;
                 doml:class     ont:Student ;
                 doml:toTable   map:person-table ;
                 doml:when      map:status-stud-cond .
```

The condition `map:status-stud-cond` will be defined later in the part of conditions and transformation specifications.

For simplification, this bridge is noted:

$\boxed{\textit{student-cb} = \textit{CB}(\textbf{Student}, \textbf{person}, \textit{WHEN}(\textit{person.status}=\textit{``Student''}))}$

In this case, the third argument (initiated with *WHEN*) indicates the associated condition.

Similarly, the third bridge relates the concept Employee to the table person with a condition which is: the column *person.status* has the value: "Employee". This bridge is encoded in RDF as follows:

```
map:employee-cb  a             doml:ConceptBridge ;
                 doml:class     ont:Employee ;
                 doml:toTable   map:person-table ;
                 doml:when      map:status-emp-cond .
```

For simplification, this bridge is noted:

$\boxed{\textit{employee-cb} = \textit{CB}(\textbf{Employee}, \textbf{person}, \textit{WHEN}(\textit{person.status}=\textit{``Employee''}))}$

The fourth concept bridge relates the concept Department with the table department. It is encoded in RDF as follows:

```
map:department-cb    a              doml:ConceptBridge ;
                     doml:class     ont:Department ;
                     doml:toTable   map:department-table ;
```

and in the simplified format as:

$\boxed{\textit{department-cb} = \textit{CB}(\textbf{Department}, \textbf{department})}$

#### 6.3.4.2 Datatype Property Bridges

Datatype Property Bridges (DPBs) are used to relate datatype properties with database columns. Each datatype property bridge has a unique identifier within a mapping document. A datatype property bridge belongs to exactly one concept bridge, called domain-concept-bridge (DCB), which is the concept bridge of the datatype property' domain.

A datatype property bridge can relates a datatype property to one or more database columns, directly or via transformations, and, possibly, with conditions.

In DOML specification, the primitive `doml:DatatypePropertyBridge` is used to represent datatype property bridge. This primitive has the following features:

- `doml:datatypeProperty` that indicates the mapped datatype property (`owl:DatatypeProperty`).
- `doml:toColumn` that indicates the corresponding database column (an instance of `doml:Column`).
- `doml:toTransform` that indicates a transformation (over one or more columns) that corresponds to the mapped datatype property. This transformation is an instance of `doml:Transformation` (see Section 6.3.5.2).
- `doml:belongsToConceptBridge` that indicates the domain-concept-bridge of the datatype property bridge (should be already defined as an instance of `doml:ConceptBridge`).
- `doml:when` indicates an optional condition that can be associated to the datatype property bridge. A conditional datatype property bridge means that this bridge is usable only when the associated condition holds.

In RDF representation of DOML, a datatype property bridge is encoded as follows:

```
[bridge-ID]     a                           doml:DatatypePropertyBridge;
                doml:datatypeProperty       [datatype-property-URI];
                doml:toColumn               [column-resource];
                doml:toTransform            [transformation];
                doml:belongsToConceptBridge [concept-bridge-id];
                doml:when                   [condition] .
```

where:

- `[bridge-ID]` is a resource representing the identifier of the datatype property bridge (an instance of `doml:DatatypePropertyBridge`).
- `[datatype-property-URI]` is the URI of the mapped datatype property.
- `[column-resource]` is a resource representing the mapped database column (an instance of `doml:Column`), this resource should be already defined in the part of database schema specification.
- `[transformation]` is a resource representing the transformation that corresponds to the mapped datatype property (an instance of `doml:Transformation`). This resource should be defined somewhere in the mapping document.
- `[concept-bridge-id]` is a resource representing the domain-concept-bridge of the datatype property bridge (an instance of `doml:ConceptBridge`), this resource should be already defined in the document.
- `[condition]` is a resource representing an optional associated condition (an instance of `doml:Condition`), this resource should be defined somewhere in the mapping document.

For simplification, a direct datatype property bridge is noted:

$\boxed{\textit{id} = \textit{DPB}(\textbf{dp}, \textbf{domBrdgId}, \textbf{col})}$

where:

- *id* is the identifier of the datatype property bridge,
- the symbol *DPB* is used to indicate that this bridge is a Datatype Property Bridge
- the first argument **dp** is the mapped datatype property.
- the second argument **domBrdgId** indicates the domain-concept-bridge.
- the third argument **col** is the mapped database column.

However, a datatype property bridge that includes a transformation is noted:

$\boxed{\textit{id} = \textit{DPB}(\textbf{dp}, \textbf{domBrdgId}, \textbf{Trans}(col_1, \cdots, col_n))}$

where the third argument $\textbf{Trans}(col_1, \cdots, col_n)$ indicates the corresponding transformation **Trans** over the database columns $col_1, \cdots, col_n$.

In both cases, an optional condition can be represented as a fourth argument: *WHEN*(*condition*).

**Example** – In our running example, we define a datatype property bridge that relates the datatype property **name** (of the concept **Person**) to a transformation (**CONCAT**) over two database columns: *firstname* and *lastname* of the table *person*. This datatype property bridge belongs to the concept bridge **person-cb** and it is noted using the simplified format as :

$\boxed{\textit{person-name-dpb} = \textit{DPB}(\textbf{name}, \textbf{person-cb}, \textbf{CONCAT}(\textit{firstname}, \textit{lastname}))}$

and it is encoded in RDF as:

```
map:person-name-dpb    a                            doml:DatatypePropertyBridge ;
                       doml:belongsToConceptBridge  map:person-cb ;
                       doml:datatypeProperty        ont:name ;
                       doml:toTransform             map:fn-ln-concat .
```

Note that `map:fn-ln-concat` represents the transformation **CONCAT**(*firstname*, *lastname*) and will be defined later (see Section 6.3.5.2).

Since the datatype property **name** is inherited by the two sub-concepts **Student** and **Employee**, we define two further datatype property bridges for this inherited property. These two additional datatype properties relate the datatype property **name** to the transformation **CONCAT**(*firstname*, *lastname*), but they belong to the concept bridges **student-cb** and **employee-cb** respectively. In addition, each of these two datatype property bridges has an associated condition. The first bridge has the condition *person.status=“Student”* (the column *person.status* has the value: *“Student”*), whereas the second has the condition *person.status=“Employee”*.

These two bridges are noted using the simplified notation as follows:

*student-name-dpb* = *DPB*(**name**, **student-cb**, **CONCAT**(*firstname*, *lastname*), *WHEN*(*person.status=“Student”*))
*employee-name-dpb* = *DPB*(**name**, **employee-cb**, **CONCAT**(*firstname*, *lastname*), *WHEN*(*person.status=“Employee”*))

and using RDF representation as:

```
map:student-name-dpb   a                            doml:DatatypePropertyBridge ;
                       doml:belongsToConceptBridge  map:student-cb ;
                       doml:datatypeProperty         ont:name ;
                       doml:toTransform              map:fn-ln-concat ;
                       doml:when                     map:status-stud-cond .

map:employee-name-dpb  a                             doml:DatatypePropertyBridge ;
                       doml:belongsToConceptBridge   map:employee-cb ;
                       doml:datatypeProperty         ont:name ;
                       doml:toTransform              map:fn-ln-concat ;
                       doml:when                     map:status-emp-cond .
```

Note that `status-stud-cond` and `status-emp-cond` represents the conditions (*person.status= "Student"*) and (*person.status= "Employee"*) respectively, and will be defined later.

#### 6.3.4.3 Object Property Bridges

Object Property Bridges (OPBs) are used to map object properties to relationships between database tables (usually, foreign-primary-keys relationships). Each object property bridge has a unique identifier within a mapping document. An object property bridge belongs to exactly one concept bridge, called domain-concept-bridge (DCB), which is the concept bridge of the object property' domain. An object property bridge also refers to a concept bridge called range-concept-bridge (RCB), which is the concept bridge of the object property' range.

Similar to other mapping bridges, object property bridges can possibly have associated conditions. However, they can't have transformations because object properties refers to instances, not to literal values (transformations apply on values).

Finally, an object property bridge has a join expression that indicates how the related tables have to be joined together.

In DOML specification, the primitive `doml:ObjectPropertyBridge` is used to represent object property bridges. This primitive has the following features:

- `doml:objectProperty` that indicates the mapped object property (`owl:ObjectProperty`).
- `doml:belongsToConceptBridge` that indicates the domain-concept-bridge of the object property bridge (should be already defined as an instance of `doml:ConceptBridge`).
- `doml:refersToConceptBridge` that indicates the range-concept-bridge of the object property bridge (should be already defined as an instance of `doml:ConceptBridge`).
- `doml:when` that indicates an optional condition that can be associated to the object property bridge. A conditional object property bridge means that this bridge is usable only when the associated condition holds.
- `doml:join-via` that indicates the relational join expression (an instance of `doml:Condition`, see Section 6.3.5.1) according to which the related tables have to be joined together.

In RDF representation of DOML, an object property bridge is encoded as follows:

```
[bridge-ID]   a                             doml:ObjectPropertyBridge;
              doml:objectProperty           [object-property-URI];
              doml:belongsToConceptBridge   [domain-cb-id];
```

```
            doml:refersToConceptBridge    [range-cb-id];
            doml:join-via                 [join-expr];
            doml:when                     [condition] .
```

where:

- `[bridge-ID]` is a resource representing the identifier of the object property bridge (an instance of `doml:ObjectPropertyBridge`).
- `[object-property-URI]` is the URI of the mapped object property.
- `[domain-cb-id]` and `[range-cb-id]` are two resources representing the domain-concept-bridge and the range-concept-bridge, respectively, of the object property bridge (both are instances of `doml:ConceptBridge`), these resources should be already defined in the document.
- `[join-expr]` is a resource representing the associated join expression (an instance of `doml:Condition`), this resource should be defined somewhere in the mapping document.
- `[condition]` is a resource representing an optional associated condition (an instance of `doml:Condition`), this resource should be defined somewhere in the mapping document.

For simplification, an object property bridge is noted:

$\boxed{\textit{id} = \textit{OPB}(\textbf{op}, \textbf{dcb-Id}, \textbf{rcb-Id}, \textit{JOIN}(\textit{join-expr}), \textit{WHEN}(\textit{condition}))}$

where:

- *id* is the identifier of the object property bridge,
- the symbol *OPB* is used to indicate that this bridge is an Object Property Bridge
- the first argument **op** is the mapped object property.
- the second argument **dcb-Id** indicates the domain-concept-bridge.
- the third argument **rcb-Id** indicates the range-concept-bridge.
- the fourth argument indicates the join expression.
- the fifth argument indicates an optional condition expression.

**Example** – In our running example, we define an object property bridge that relates the object property **studies-in** (from the concept **Student** to the concept **Employee**) to the relationship between the tables **Person** and **Department** (via the join expression *person.dept_id=department.dept_id*). This object property bridge belongs to the concept bridge **student-cb** and refers to the concept bridge **department-cb**. This bridge has the condition *person.status="Student"* (the column *person.status* has the value: *"Student"*).

This bridge is noted using the simplified format as :

$\boxed{\begin{array}{l}\textit{studies-in-opb} = \textit{OPB}(\textbf{studies-in}, \textbf{student-cb}, \textbf{department-cb}, \\ \textit{JOIN}(\textit{person.dept\_id=department.dept\_id}), \textit{WHEN}(\textit{person.status="Student"}))\end{array}}$

and it is encoded in RDF as:

```
map:studies-in-opb  a                           doml:ObjectPropertyBridge ;
                    doml:objectProperty         ont:studies-in ;
                    doml:belongsToConceptBridge map:student-cb ;
                    doml:refersToConceptBridge  map:department-cb ;
                    doml:join-via               map:person-dept-join ;
                    doml:when                   map:status-stud-cond .
```

Note that `map:person-dept-join` represents the join expression (*person.dept_id= department.dept_id*) and should be defined somewhere in the document.

### 6.3.5 Describing Transformations and Conditions in DOML

DOML provides several primitives that allows the specification of conditions and transformations associated with mapping bridges. The mechanism used for defining conditions and transformations is, to some extent, borrowed from R2O language.

#### 6.3.5.1 Conditions

A conditional mapping bridge is a bridge associated with a condition over the mapped database or ontology. A mapping bridge is usable (in query resolution and instance retrieval) only when the associated condition holds.

A condition can be either a basic condition or a complex condition expression. A basic condition is a simple condition statement that uses a primitive condition (such as **equals**) to relate two or more arguments. Table 6.1 shows the list of primitive conditions used in DOML language. A complex condition is a logical expression that uses a logical operator (AND, OR, or NOT) to relate other (basic and/or complex) conditions.

In DOML specification, both basic and complex conditions are represented using the primitive `doml:Condition`. This primitive has the following features:

- `doml:conditionType` – If the condition is basic, this feature indicates the condition type (an instance of `doml:PrimitiveCondition`), such as `doml:Equals`, `doml:Hi-than`, etc. (see Table 6.1). If the condition is complex, this feature indicates a logical operator: `AND`, `OR`, and `NOT`.
- `doml:hasArguments` – This feature indicates the ordered list of arguments of this condition (an instance of `doml:ArgList`, see Section 6.3.5.3).

In RDF representation of DOML, a condition is encoded as follows:

```
[condition-ID] a                  doml:Condition;
               doml:conditionType [operator];
               doml:hasArguments  [arg-list].
```

where:

- `[condition-ID]` is a resource representing the identifier of the condition (an instance of `doml:Condition`).
- `[operator]` is either the condition type (an instance of `doml:PrimitiveCondition`) if the condition is basic, or a logical operator (one of `doml:AND`, `doml:OR` and `doml:NOT`) if the condition is complex.
- `[arg-list]` is the resource representing the list of arguments of the condition (an instance of `doml:ArgList`). This resource should be defined somewhere in the document, and can be reused by other conditions or transformations.

**Example** – In our running example, we define two basic conditions to determine the status of a person (either a student or an employee): (*person.status= "Student"*), (*person.status= "Employee"*). These conditions are encoded in RDF as follows:

```
map:status-stud-cond a              doml:Condition ;
                doml:conditionType doml:Equals-str ;
                doml:hasArguments  map:arg-list12.
map:status-emp-cond  a              doml:Condition ;
                doml:conditionType doml:Equals-str ;
                doml:hasArguments  map:arg-list13.
map:arg-list12  a        doml:ArgList ;
                rdf:_1   map:person-status-col;
                rdf:_2   "Student".
map:arg-list13  a        doml:ArgList ;
                rdf:_1   map:person-status-col;
                rdf:_2   "Employee".
```

TABLE 6.1: Primitive conditions, transformations, and logical operators used in DOML language

| Primitive Transformations | Primitive Conditions | Logical Operators |
|---|---|---|
| `doml:Get-nth-char` | `doml:Equals` | `doml:AND` |
| `doml:Get-substring` | `doml:Equals-str` | `doml:OR` |
| `doml:Get-indexof` | `doml:Lo-than` | `doml:NOT` |
| `doml:Get-string-length` | `doml:Lo-than-str` | |
| `doml:Concat` | `doml:Hi-than` | |
| `doml:Add` | `doml:Hi-than-str` | |
| `doml:Subtract` | `doml:In-keyword` | |
| `doml:Multiply` | `doml:Between` | |
| `doml:Divide` | `doml:Between-str` | |
| | `doml:Date-before` | |
| | `doml:Date-after` | |
| | `doml:Date-equal` | |

#### 6.3.5.2 Transformations

Transformations are used to define sophisticated relations between values within the mapped database and ontology. A transformation can be applied on one or more arguments, that can refer to constant values, to database columns, or to other transformations. Transformations are particularly used within datatype property bridges and within conditions.

DOML provides a set of primitive transformations (borrowed from R2O language) that can be used to define concrete transformations inside a mapping document. These primitive transformations are listed in Table 6.1.

In DOML specification, the primitive `doml:Transformation` is used to represent transformations. This primitive has the following features:

- `doml:transType` that indicates the type of this particular transformation operation (an instance of `doml:PrimitiveTransformation`), which can be one of those shown in Table 6.1.

- `doml:hasArguments` that indicates the ordered list of arguments of this transformation (an instance of `doml:ArgList`, see Section 6.3.5.3).

In RDF representation of DOML, a transformation is encoded as follows:

```
[transform-ID]    a                  doml:Transformation;
                  doml:transType     [primitive-transformation];
                  doml:hasArguments  [arg-list].
```

where:

- `[transform-ID]` is a resource representing the identifier of the transformation (an instance of `doml:Transformation`).
- `[primitive-transformation]` is the URI of the used primitive transformation (an instance of `doml:PrimitiveTransformation`), that can be one of those listed in Table 6.1.
- `[arg-list]` is the resource representing the list of arguments of the transformation (an instance of `doml:ArgList`). This resource should be defined somewhere in the document, and can be reused by other transformations or conditions.

In our running example, we define a transformation that is used to concatenate two database columns representing the first and the last name of a person (*person.firstName* and *person.lastname*) giving a full name. This transformation has the transformation type: `doml:Concat`, and two arguments (grouped in a list) that refer to the column resources (`person-firstname-col` and `person-lastname-col`) of the respective columns.

This transformation is encoded in RDF as follows:

```
map:fn-ln-concat a                  doml:Transformation ;
                 doml:transType     doml:Concat ;
                 doml:hasArguments  map:arglist-fnln.
map:arglist-fnln a         doml:ArgList;
                 rdf:_1    map:person-firstname-col;
                 rdf:_2    map:person-lastname-col .
```

#### 6.3.5.3 Arguments

Both conditions and transformations have arguments. An argument can refer to a constant value, to a database column, or to a transformation.

The arguments of conditions and transformations are represented using argument lists. In DOML specification, the primitive `doml:ArgList` is used to represent an ordered list of the arguments of a transformation or condition. The order of arguments is specified using `rdf:_1`, `rdf:_2`, $\cdots$ predicates. An argument can be

- a constant literal value,
- a database column (an instance of `doml:Column`),
- a transformation (an instance of `doml:Transformation`), or
- a condition (an instance of `doml:Condition`).

In RDF representation of DOML, an argument list is encoded as follows:

```
[arglist-ID]    a         doml:ArgList;
                 rdf:_1    [param1] ;
                 rdf:_2    [param2] ;
                 ...
```

where:

- `[arglist-ID]` is a resource representing the identifier of this argument list (an instance of doml:ArgList).
- `[param1]`, `[param2]`, ⋯ : are resources or literal values representing the respective parameters. A parameter can be:
  - a constant literal value (string, integer, decimal, date, ⋯)
  - a resource representing a database column (an instance of `doml:Column`).
  - a resource representing a transformation (an instance of `doml:Transformation`), or
  - a resource representing a condition (an instance of `doml:Condition`).

Resources representing columns, transformations and conditions should be defined somewhere in the mapping document.

Argument lists can be used/reused by multiple conditions and/or transformations.

## Chapter Summary

In this chapter, we have proposed two database-to-ontology mapping specifications: associations with SQL statements, and DOML language.

In section 6.1, we give a brief introduction to these specifications.

In Section 6.2, we presented the first mapping specification: *associations with SQL statements*. In this spefication, mappings are expressed as:

1. Concept Associations (Section 6.2.1),
2. Object Property Associations (Section 6.2.2), and
3. Datatype property Associations (Section 6.2.3)

In Section 6.3, we presented the second mapping specification: *DOML language*. We firstly motivated the need to this language (Section 6.3.1), and gave a general description of it (Section 6.3.2). In Section 6.3.3, we explained how to describe a relational schema using DOML language. The components of such a description are intended to be used within the description of mapping bridges. Then, we have presented how to specify mapping bridges in DOML language, namely, concept bridges (Section 6.3.4.1), datatype property bridges (Section 6.3.4.2), and object property bridges (Section 6.3.4.3). Finally, in Section 6.3.5, we have presented how to specify conditions and transformations in DOML language.

In the next chapter, we will introduce our database-to-ontology mapping tool, called DB2OWL. This tool supports both of our mapping specifications proposed in this chapter.

Chapter 7

# DB2OWL Tool

Contents

## Abstract

In this chapter, we present DB2OWL, our tool for database-to-ontology mapping. This tool is a part of the data provider module within OWSCIS architecture. It is intended to wrap a local database to a local ontology at the site level.

DB2OWL tool has three main tasks:

1. generate an ontology from a relational database,
2. map a relational database to an existing OWL ontology, and
3. translate SPARQL queries over an ontology into SQL queries over a mapped database.

Firstly, we give a general description of DB2OWL (Section 7.2). This tool supports two kinds of database-to-ontology mapping specification. The first kind is called: "*Associations with SQL Statements*". This kind of mapping specification is described in Section 6.2. The second kind is DOML mapping language which has been already presented in Section 6.3.

In Section 7.3, we present in details the first task of DB2OWL which is: *ontology generation from a database*. Then, in Section 7.4, we present the second task of DB2OWL which is: mapping a database to an existing ontology. The third task of DB2OWL tool, which is SPARQL-to-SQL query translation, is presented in Chapter 8. Finally, in Section 7.5, we give some details about the implementation of a prototype of DB2OWL.

## 7.1 Motivation

As mentioned in Section 3.3, in OWSCIS cooperation system, a participant site can contain several information sources of different types. In particular, relational databases and XML data sources are supported. A data provider is a module that encapsulates the set of information sources located at a single site. The main purpose of a data provider is to wrap local information sources within a site to a local ontology. A data provider contains the following components:

- Local ontology – which represents the semantics of local information sources.
- Mappings between the local and the global ontologies.
- Mappings between the local ontology and local information sources.

A participating site may already have its own local ontology. In this case, each local information source needs to be mapped to this existing local ontology. The result of this process is a mapping document that describes the correspondences between the information source and the local ontology.

However, the local ontology does not necessarily exist before the connection to OWSCIS platform. In this case, the local ontology has to be created from the information sources. Therefore, there is a need to a methodology for ontology generation from information sources. Such methodology depends on the number and the type of information sources. We distinguish three cases according to the information sources contained in a site:

- The site contains a single database.

- The site contains a single XML data source.
- The site contains multiple databases and/or XML data sources.

For each of these cases, we need a specific methodology for ontology creation. We need also a mapping document that describes the correspondences between the generated ontology and each information source.

In summary, a methodology is needed for the following tasks:

1. Ontology creation from a single database
2. Ontology creation from a single XML data source
3. Ontology creation from multiple databases and XML data sources
4. Mapping a database to an existing ontology
5. Mapping an XML data source to an existing ontology

In the first case where a site contains a single database, we need a tool that handle the tasks of 1) ontology creation from this single database, and 2) mapping this single database to an existing ontology. We realized this tool and we call it DB2OWL.

## 7.2 DB2OWL Description

As we mentioned previously, a data provider should perform three fundamental tasks:

1. Create the local ontology.
2. Map information sources to the local ontology.
3. Process local queries.

DB2OWL is a tool implemented to handle these tasks within a data provider for the case of a single database. That is, DB2OWL tool has three main functionalities:

1. Create an ontology from a single database.
2. Map a single database to an existing ontology.
3. Process queries over ontology towards queries over a single database.

The first and second tasks will be presented in this chapter, whereas the third task will be presented in the next chapter.

### 7.2.1 Creation of the Local Ontology from a Single Database

The first goal of DB2OWL tool is to automatically create an ontology from a relational database. The created ontology plays the role of the local ontology within the data provider, and is described in OWL-DL language.

The ontology generation process starts by detecting some particular cases for tables in the database schema. According to these cases, each database component (table, column, constraint) is then converted to a corresponding ontology component (class, property, relation).

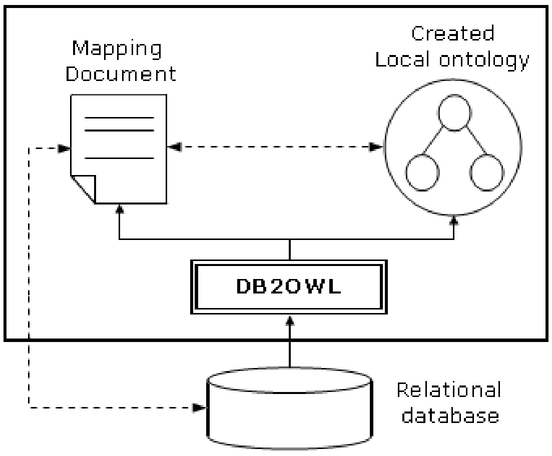


FIGURE 7.1: DB2OWL: ontology creation from a single database

Since we use a non-materialized approach, the generated ontology only contains the description of the concepts and properties but not the instances. Data instances are retrieved and translated as needed in response to user queries. During ontology generation process, DB2OWL also generates a mapping document that describes the correspondences between the database and the generated local ontology. The mapping document is used for query processing purposes.

### 7.2.2 Mapping a Single Database to the Local Ontology

A participating site may already have its own local ontology, and does not want to create a new one from the database. In this case, the database should be mapped to this existing local ontology. DB2OWL allows a human expert to manually indicates the mappings between the database and the local ontology. The result of this process is a mapping document that describes the correspondences between the database and the local ontology.

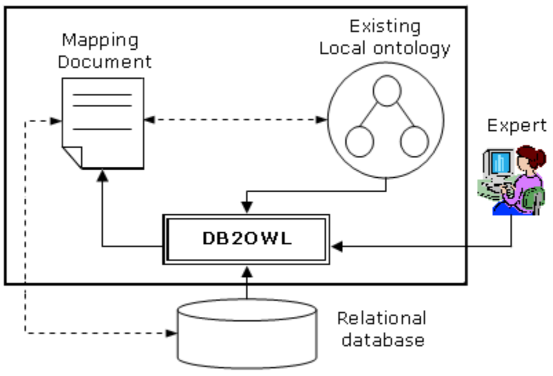


FIGURE 7.2: DB2OWL: mapping a single database to the local ontology

In both of these tasks (ontology creation from a database, and mapping a database to an existing ontology), a mapping document is produced as a result of the mapping process. This mapping document describes the correspondences between the entities of the database and the ontology. These correspondences can be specified using two kinds of mapping specifications that we presented in the previous chapter: associations with SQL statements and DOML language.

In the reminder of this chapter, we introduce the tasks of DB2OWL tool. We start by the task of ontology generation from a database.

## 7.3 Ontology Generation from a Database

The first goal of DB2OWL tool is to automatically create an ontology from a relational database. The created ontology is described in OWL-DL language. Since we use a non-materialized approach, the generated ontology is instance-free and only contains the concepts and properties.

The ontology generation process starts by detecting some particular cases for tables in the database schema. According to these cases, each database component (table, column, constraint) is then converted to a corresponding ontology component (concept, property, etc.). During the ontology generation process, DB2OWL also generates a mapping document that describes the correspondences between the database and the generated local ontology. This mapping document can be expressed using two kinds of mapping specifications: Associations with SQL statements and DOML.

In the following subsections, we introduce some notations that we use to describe the database metadata. Then, we explain the table cases that should be detected in the database in order to exploit them throughout the ontology generation process. The ontology generation process itself is then introduced. Finally we illustrate the mechanism of mapping generation during the process.

### 7.3.1 Running Example

In this subsection, we introduce a simple database as a running example. This example will be used throughout the following subsections in order to illustrate the notations, and the different steps of ontology generation process and mapping generation process.

This example represents a university database. There are several diplomas, each consisting of several modules. A student can be enrolled in (at most) one diploma. Each module is taught through a series of sessions. For each session, we record the hall, the time and the lecturer. To observe the presence of students at the sessions, we have a junction table `Presence` that associates a session with attendee students. The complete database schema is shown in Figure 7.3.

### 7.3.2 Notations

In this section, we introduce some notations that we use to describe the database metadata. These notations will be used later to illustrate the particular cases of database tables.

Let $DB$ be a database and let $T$ be a table of DB, we denote the set of $T$ columns as $COL(T)$, the set of $T$ primary keys as $P(T)$, and the set of $T$ foreign keys as $F(T)$.

We use $PF(T)$ to denote the set of $T$ columns which are both primary and foreign keys, $P\_(T)$ to denote the set of $T$ columns which are primary but not foreign keys, $\_F(T)$ denote the set of $T$ columns which are foreign but not primary keys, and $\_\_(T)$ to denote the set of $T$ columns which are neither primary nor foreign keys.

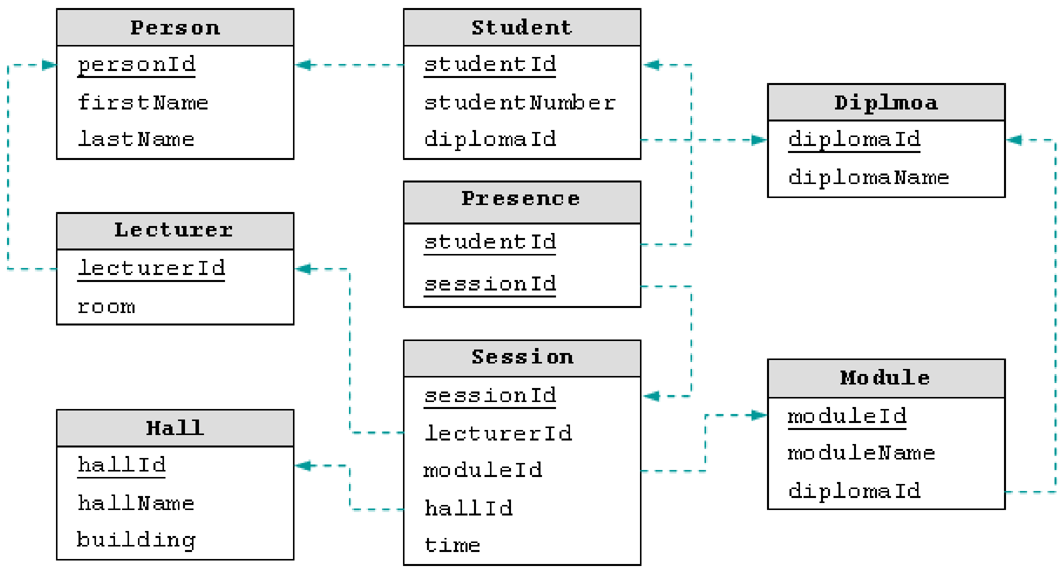


FIGURE 7.3: Running example

We can note that:

$$PF(T) = P(T) \cap F(T)$$
$$PF(T) \cup P\_(T) = P(T)$$
$$PF(T) \cup \_F(T) = F(T)$$

The sets $PF(T)$, $P\_(T)$, $\_F(T)$, $\_\_(T)$ are a partition of $COL(T)$, that is, the union of these sets is equal to $COL(T)$, and they are pairwise disjoint.

Let us show how to use these notations with the table Student of our running example:

$COL(Student) = \{studentId, studentNumber, diplomaId\}$
$P(Student) = \{studentId\}$
$F(Student) = \{studentId, diplomaId\}$
$PF(Student) = \{studentId\}$
$P\_(Student) = \{\}$
$\_F(Student) = \{diplomaId\}$
$\_\_(Student) = \{studentNumber\}$

**Referential integrity constraints**

A referential integrity constraint has the form:

$ric((T_1, A_1), (T_2, A_2))$

where:

- $T_1, T_2 \in DB$ are database tables,
- $A_1 = \{c_{11}, c_{12}, \cdots\} \subseteq COL(T_1)$ and $A_2 = \{c_{21}, c_{22}, \cdots\} \subseteq COL(T_2)$ are subsets of $T_1$ and $T_2$ columns respectively,
- $|A_1| = |A_2|$,
- each column $c_1^i \in A_1$ is a foreign key references a primary key $c_2^i \in A_2$, this means that $A_1 \subseteq F(T_1)$ and $A_2 \subseteq P(T_2)$.

For a referential integrity constraint $ric((T_1, A_1), (T_2, A_2))$, we define the following functions:

- local table ($LT$), returns the table owning this referential integrity constraint, $LT(ric) = T_1$
- local attributes ($LA$), returns the columns (actually, foreign keys) of $T_1$ composing the constraint, $LA(ric) = A_1$
- reference table ($RT$), returns the table referenced by this referential integrity constraint, $RT(ric) = T_2$
- reference attributes ($LA$), returns the columns (actually, primary keys) of $T_2$ referenced by the constraint, $RA(ric) = A_2$

Let $\mathcal{RIC}$ be the set of all explicit referential constraints in a database $DB$, we also define the function $RIC$ from $DB$ to $\mathcal{RIC}$. This function returns for each table $T \in DB$, the set of referential integrity constraints whose local table is $T$. That is,

$RIC : DB \to 2^{\mathcal{RIC}}, RIC(T) = \{ric((T_1, A_1), (T_2, A_2)) \in \mathcal{RIC} : LT(ric) = T\}$.

For example, we consider the table `Student` of our running example. This table has two foreign keys: 1) `studentId` which references `Person.personId`, and 2) `diplomaId` which references `Diploma.diplomaId`. Thus, `Student` table has two referential integrity constraints:

$$\begin{aligned} RIC(Student) = \; & \{ric_1((Student, \{studentId\}), (Person, \{personId\})), \\ & ric_2((Student, \{diplomaId\}), (Diploma, \{diplomaId\}))\} \end{aligned}$$

The results of the functions: local table, local attributes, reference table and reference attributes for the first constraint are:

$$\begin{aligned} LT(ric_1) &= Student & LA(ric_1) &= studentId \\ RT(ric_1) &= Person & RA(ric_1) &= personId \end{aligned}$$

and for the second constraints are:

$$\begin{aligned} LT(ric_2) &= Student & LA(ric_2) &= diplomaId \\ RT(ric_2) &= Diploma & RA(ric_2) &= diplomaId \end{aligned}$$

### 7.3.3 Particular Table Cases

The ontology generation process used in our approach relies on particular cases for database table. These cases are taken into account during the ontology creation. These cases will be illustrated using our running example database.

#### Case 1

When a table $T$ is used as junction table to relate two other tables $T_1$, $T_2$ in a many-to-many relationship, it can be divided into two disjoint subsets of columns $A_1$, $A_2$, each participating in a referential constraint with $T_1$ and $T_2$ respectively:

$RIC(T) = \{ric_1, ric_2\} : ric_1((T, A_1, ), (T_1, P(T_1))), ric_2((T, A_2), (T_2, P(T_2)))$

In this case the primary key for $T$ is formed from the two foreign keys (i.e. copies of the primary keys of $T_1$ and $T_2$). Therefore, all $T$ columns are both primary and foreign keys, i.e., $COL(T) = F(T) = P(T)$, therefore, $COL(T) = PF(T)$. Thus, the necessary and sufficient condition for a table $T$ to be in **Case 1** is:

$$COL(T) = PF(T) \text{ and } |RIC(T)| = 2$$

**Example**

Let us consider the table `Presence` in our running example. This table consists of two columns $COL(Presence) = \{studentId, sessionId\}$.

We note that $P(Presence) = \{studentId, sessionId\}$ and $F(Presence) = \{studentId, sessionId\}$, therefore, $PF(Presence) = \{studentId, sessionId\} = COL(Presence)$. In addition, $RIC(T) = \{ric_1, ric_2\}$ where:

$$ric_1((Presence, \{studentId\}), (Student, \{studentId\}))$$
$$ric_2((Presence, \{sessionId\}), (Session, \{sessionId\}))$$

thus, $|RIC(T)| = 2$, therefore `Presence` is in **Case 1**.

### Case 2

This case occurs when a table $T$ is related to another table $T_1$ by a referential integrity constraint whose local attributes are also primary keys:

$$\exists ric \in RIC(T) : LA(ric) = P(T)$$

In other words: $ric(T, P(T), T_1, P(T_1))$ In this case all the primary keys of $T$ are foreign keys because they participate in a referential integrity constraint: $P\_(T) = \phi$ . Thus, the necessary and sufficient condition for a table $T$ to be in **Case 2** is:

$$\exists ric \in RIC(T) : LA(ric) = P(T)$$

**Example**

Let us consider the table `Lecturer` of our running example. This table consists of two columns $COL(Lecturer) = \{lecturerId, room\}$. We find that $P(Lecturer) = \{lecturerId\}$ and $RIC(T) = \{ric_1\}$ where: $ric_1((Lecturer, \{lecturerId\}), (Person, \{personId\}))$ We note that $LA(ric_1) = \{lecturerId\} = P(Lecturer)$, therefore, `Lecturer` is in **Case 2**.

### Case 3

This case is the default case, it occurs when none of previous cases occur.

**Example**

Let us consider the table `Hall`, it consists of the columns

$COL(Hall) = \{hallId, hallName, building\}$.

We note that $P(Hall) = \{hallId\}$ and $F(Hall) = \{\}$, therefore $PF(Hall) = \{\}$. This means that `Hall` is not in **Case 1**. At the other hand, we note Hall has no referential integrity constraints: $RIC(T) = \{\}$. Therefore, Hall can not be in **Case 2**. Consequently, since Hall neither in **Case 1** nor in **Case 2**, then it is in **Case 3**.

The different cases are summarized in Table 7.1.

As previously mentioned, these particular cases of tables are firstly detected in the database. Then, the ontology generation process can use them to appropriately map database components to suitable ontology components as we will see in the next section.

| Case | Necessary and sufficient condition | Example |
|---|---|---|
| **Case 1** | $COL(T) = PF(T)$ and $\|RIC(T)\| = 2$ | `Presence` |
| **Case 2** | $\exists ric \in RIC(T) : LA(ric) = P(T)$ | `Lecturer`, `Student` |
| **Case 3** | $T$ is not in **Case 1** nor in **Case 2** | All the other tables |

TABLE 7.1: The different particular cases used in mapping process

### 7.3.4 Ontology Generation Process

The ontology generation process is performed progressively. It starts by transforming tables to concepts, and then transforming columns to properties. Thus, the table cases mentioned above are used twice: one time for table-to-class transformation and then for column-to-property transformation. The ontology generation process consists of the following steps:

**Step 1**

Each database table that is in **Case 3** is transformed into OWL class. In our running example, the tables `Person`, `Session`, `Hall`, `Diploma`, and `Module` are in **Case 3**. Therefore, an OWL class is created for each one of these tables (Figure 7.4).

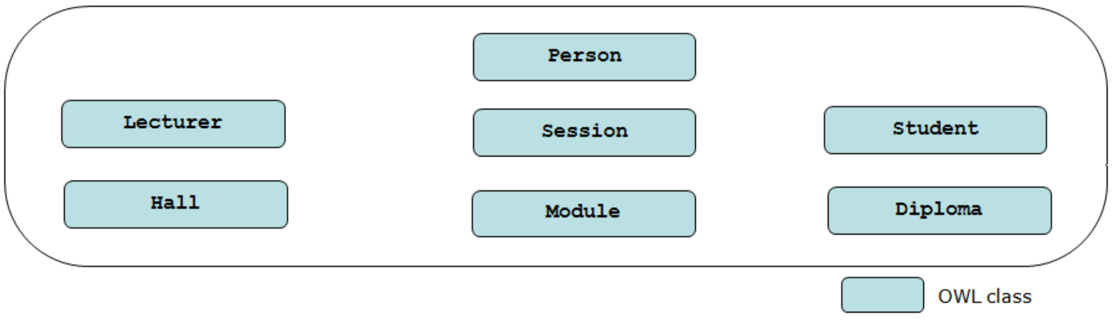


FIGURE 7.4: The result of Step 1 of ontology generation process

**Step 2**

Each table which is in **Case 2** is transformed into OWL class, and it is set as subclass of the class corresponding to its related table. That is, if $T$ is in **Case 2**, then there is a referential integrity constraint $ric \in RIC(T)$ where $ric(T, P(T), T_1, P(T_1))$. Thus, $T$ is transformed to a subclass of the class corresponding to $T_1$.

In our running example, the tables `Student` and `Lecturer` are both in **Case 2**, because each of them is related to `Person` table using a foreign key which is also a primary key (`studentId` in `Student`, and `lecturerId` in `Lecturer`). Consequently, these tables are transformed into subclasses of the class corresponding to the table `Person` (Figure 7.5).

**Step 3**

Each table which is in **Case 1** is not transformed into OWL class, but the many-to-many relationship that it represents is expressed using object properties. Two inverse object properties are added to the ontology, one for each class whose corresponding table was related to the current table. In other words, when a table $T$ is in **Case 1** then there are two referential integrity constraints: $ric_1(T, A_1, T_1, P(T_1))$ and $ric_2(T, A_2, T_2, P(T_2))$. Let us consider $C_1$ and $C_2$ are the

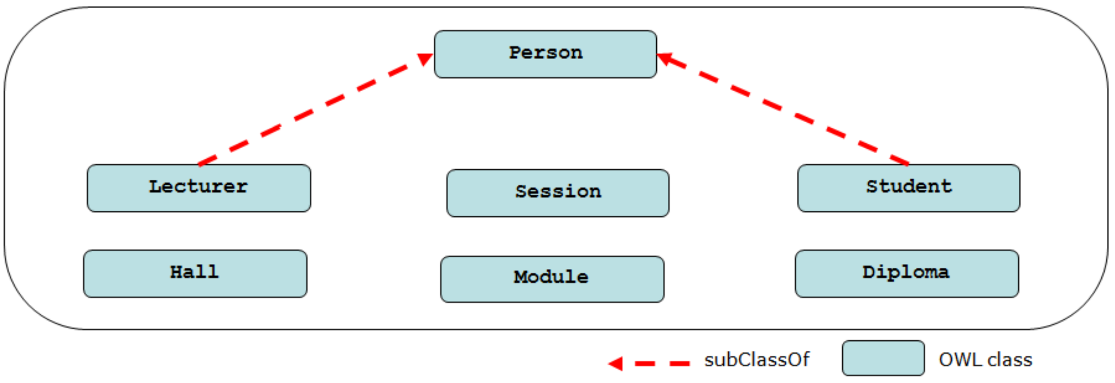


FIGURE 7.5: The result of Step 2 of ontology generation process

two classes corresponding to $T_1$ and $T_2$ respectively, so we assign to $C_1$ an object property $op_1$ whose range is $C_2$ , and assign to $C_2$ an object property $op_2$ whose range is $C_1$. These two object properties ($op_1$ and $op_2$) are inverse.

In our running example, the table `Presence` is in **Case 1** because it relates two other tables `Student` and `Session` in a many-to-many relationship. Therefore, `Presence` is not transformed to OWL class, but we assign to the class `Student` an object property `presence-session` whose range is the class `Session`, and we assign to the class `Session` an object property `presence-student` whose range is the class `Student` (Figure 7.6).

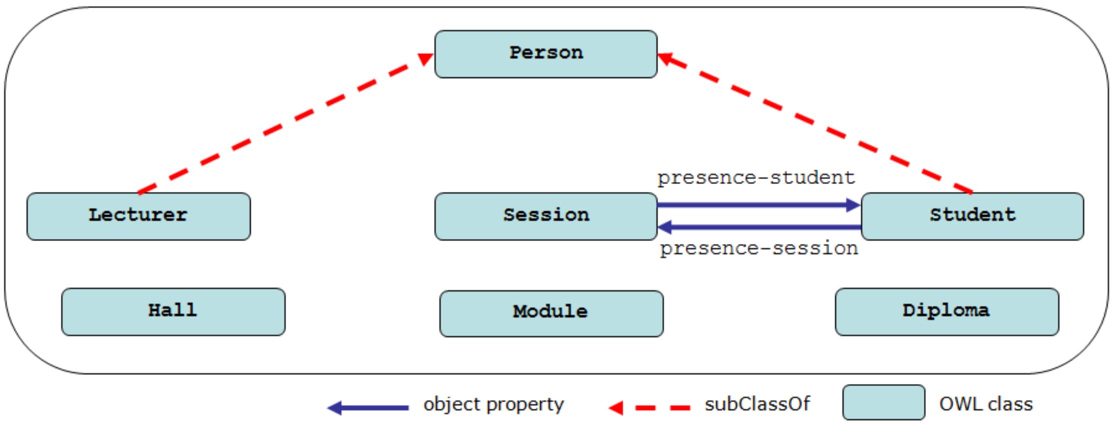


FIGURE 7.6: The result of Step 3 of ontology generation process

### Step 4

For tables that are in **Case 3**, each referential integrity constraint is transformed into an object property. Such object property will have as range the class corresponding to the table referenced by that constraint. That is, if a table $T$ is in **Case 3** and has a referential integrity constraint $ric(T, A, T_1, A_1)$, and if $C$ and $C_1$ are the classes corresponding to $T$ and $T_1$ respectively, then we assign to $C$ an object property $op$ whose range is $C_1$, and we assign to $C_1$ an object property $op'$ (inverse of $op$) whose range is $C$. To preserve the original direction of the referential integrity constraint from $T$ to $T_1$, we set the object property $op$ as functional. This means that it will have at most one value for the same instance. This characteristic is obvious because it comes from the uniqueness of key.

In our running example, the table `Module` is in **Case 3**, and it has a referential integrity constraint with the table `Diploma`. We assign to the class corresponding to `Module` an object property `module-diploma` whose range is the class corresponding to table `Diploma`. This object property is set as functional. In addition, we assign to the class corresponding to `Diploma` an object property `diploma-module` whose range is the class corresponding to `Module`. These properties `module-diploma` and `diploma-module` are inverse (Figure 7.7).

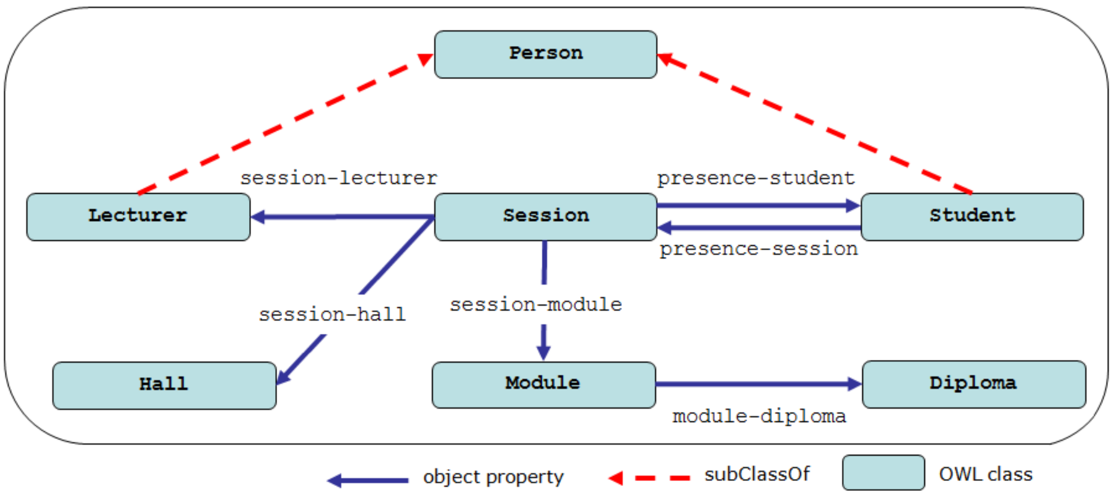


FIGURE 7.7: The result of Step 4 of ontology generation process

**Step 5**

For tables that are in **Case 2**, each referential integrity constraint, other than the constraint used in **Step 2** to create the subclass, is transformed into an object property as in the previous step.

In our running example, the table `Student` is in **Case 2**. It has two referential integrity constraints:

$$ric_1((Student, \{studentId\}), (Person, \{personId\}))$$
$$ric_2((Student, \{diplomaId\}), (Diploma, \{diplomaId\}))$$

The first one is already used in **Step 2** when we created the class corresponding to `Student` as a subclass of the class corresponding to `Person`. The second one is now used in **Step 5**. We assign an object property `student-diploma` to the class corresponding to `Student`. This property is functional and its range is the class corresponding to `Diploma`. In addition, we assign to the class corresponding to `Diploma` an object property `diploma-student` whose range is the class corresponding to `Student`. The properties `student-diploma` and `diploma-student` are inverse (Figure 7.8).

**Step 6**

Finally, for all tables, any non-key column is transformed into a datatype property (thus, tables of **Case 1** are obviously excepted from this step). The range of such datatype property is the XML schema datatype [22] equivalent to the datatype of its original column.

In our running example, the column `studentNumber` in the table `Student` is non-key, thus, it is transformed into a datatype property `student-studentNumber` whose range is XSD string datatype: `xsd:string`.

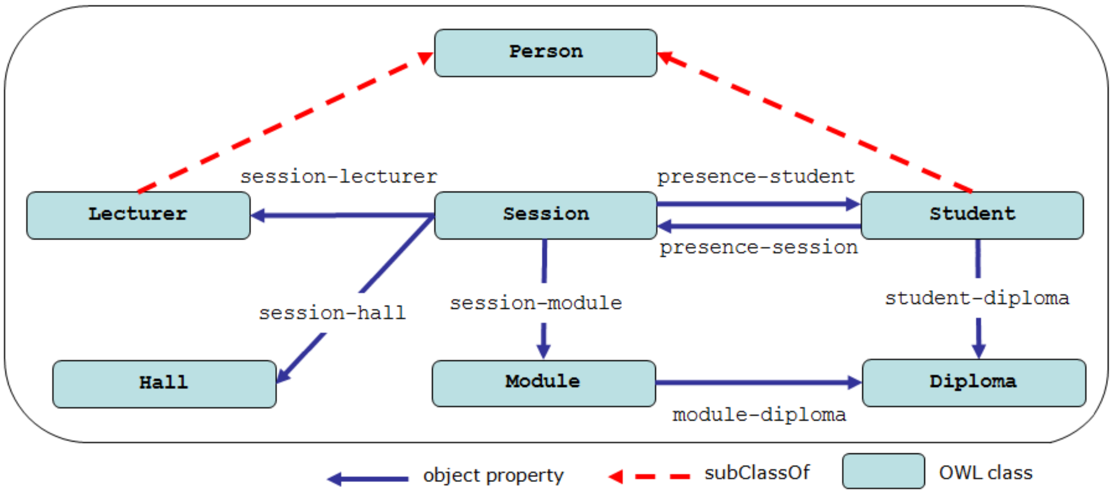


FIGURE 7.8: The result of Step 5 of ontology generation process

Figure 7.9 shows the final ontology resulting from the ontology generation process. The representation of this ontology in OWL language is shown in Appendix E.

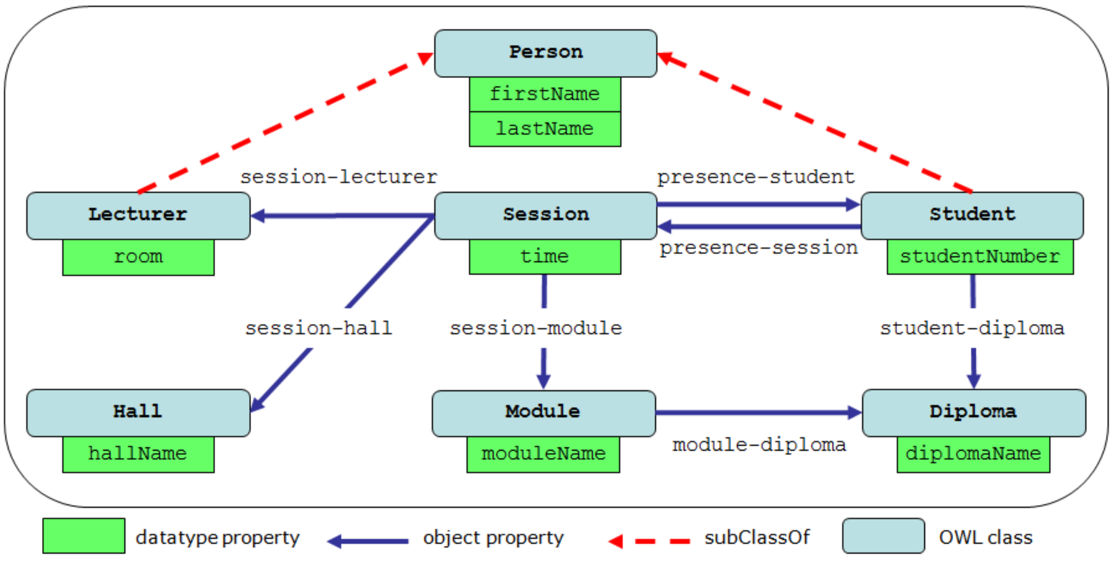


FIGURE 7.9: The final OWL ontology resulting of ontology generation process

In the next section, we will see how mappings, between the components of the created ontology and the database, are generated.

### 7.3.5 Mapping Generation Process

As we previously mentioned, during the ontology generation process, DB2OWL also generates a mapping document that describes the correspondences between the database and the generated local ontology. As we saw in Chapter 6, DB2OWL supports two kinds of mapping specifications: 1) Associations with SQL Statements (Section 6.2), and 2) DOML (6.3).

In this section, we present how to generate mappings (during the ontology generation process) using each of these two mapping specifications. We firstly explain how to generate mappings using *Associations with SQL statements* specification (Section 7.3.5.1). Then, we explain how to generate mappings using DOML language (Section 7.3.5.2). In these subsections, we represent ontology terms as qualified names (QNames) using a prefix "`ex`".

#### 7.3.5.1 Mapping Generation using Associations with SQL Statements

**Step 1**

In this step, an OWL class is generated from each database table in **Case 3**. A mapping between each class $C$ and its original table $T$ is expressed as a concept association that associates this class $C$ with an SQL statement that retrieves the primary key of the corresponding table $T$ given an alias "`DOM`". For example, the concept association for `Person` class is:

```
ex:Person ⇝ SELECT personId AS DOM FROM Person
```

**Step 2**

In this step, an OWL class $C$ is generated from database table $T$ in **Case 2**, and it is set as subclass of the class $C_1$ corresponding to the referenced table $T_1$. A mapping between the class $C$ and the table $T$ is expressed as a concept association that associates this class with an SQL statement that retrieves the primary key of the table $T$ given an alias "`DOM`". In this SQL statement, the table $T$ is joined with the reference table $T_1$ using the appropriate referential integrity constraint (actually, the constraint that let $T$ be in **Case 2**).

For example, the concept association for `Student` class is:

```
 ex:Student ⇝   SELECT studentId AS DOM FROM Student, Person
                WHERE Student.studentId = Person.personId
```

**Step 3**

In this step, there is a table $T$ which is in **Case 1** and has two referential integrity constraints: $ric_1(T, A_1, T_1, P(T_1))$ and $ric_2(T, A_2, T_2, P(T_2))$. Let us consider that $C_1$ and $C_2$ are the two classes corresponding to $T_1$ and $T_2$ respectively. In this step, no OWL class is generated, but we assign to $C_1$ an object property $op_1$ whose range is $C_2$, and assign to $C_2$ an object property op2 whose range is $C_1$. These two object properties ($op_1$ and $op_2$) are inverse. For these two object properties, we use two object property associations $OPA_1$ and $OPA_2$:

- $OPA_1$ associates to $op_1$ an SQL statement that has two items in the SELECT clause. The first item is the identifier of the domain class $C_1$ which is the primary key of $T_1$, this item has an alias "`DOM`". The second item is the identifier of the range class $C_2$ which is the primary key of $T_2$, this item has an alias "`RNG`". In this SQL statement the table $T$ is joined with both $T_1$ and $T_2$ tables using the referential integrity constraints $ric_1$ and $ric_2$ respectively. That is, the WHERE clause of the SQL statement contains the expressions: $T.pfk_1 = T_1.pk$ and $T.pfk_2 = T_2.pk$ where $A_1 = \{pfk_1\}$, $A_2 = \{pfk_2\}$, $T_1.pk \in P(T_1)$ , and $T_2.pk \in P(T_2)$.
- $OPA_2$ that associates to $op_2$ an SQL statement that has two items in the SELECT clause. The first item is the identifier of the domain class $C_2$ which is the primary key of $T_2$, this item has an alias "`DOM`". The second item is the identifier of the range class $C_1$ which is the primary key of $T_1$, this item has an alias "`RNG`". This SQL statement has the same join expression mentioned in $OPA_1$.

For example, the object property associations for `presence-session` and `presence-student` object properties are:

```
ex:presence-session ⇝
            SELECT Student.studentId AS DOM, Session.sessionId AS RNG
            FROM Student, Session, Presence
            WHERE Presence.studentId = Student.studentId
            AND Presence.sessionId = Session.sessionId
ex:presence-student ⇝
            SELECT Session.sessionId AS DOM, Student.studentId AS RNG
            FROM Session, Student, Presence
            WHERE Presence.studentId = Student.studentId
            AND Presence.sessionId = Session.sessionId
```

**Step 4**

In this step, referential integrity constraints of tables that are in **Case 3** are transformed into object properties. When a table $T$ is in **Case 3**, and has a referential integrity constraint $ric(T, A, T_1, A_1)$, and considering that $C$ and $C_1$ are the classes corresponding to $T$ and $T_1$ respectively, then we assign to $C$ an object property *op* whose range is $C_1$, and we assign to $C_1$ an object property *op*′ (inverse of *op*) whose range is $C$. For these two object properties *op* and *op*′, we use two object property associations $OPA$ and $OPA'$:

- $OPA$ associates to *op* an SQL statement that has two items in the SELECT clause. The first item is the identifier of the domain class $C$ which is the primary key of $T$, this item has an alias “`DOM`”. The second item is the identifier of the range class $C_1$ which is the primary key of $T_1$, this item has an alias “`RNG`”. In this SQL statement the table $T$ is joined with $T_1$ table using the referential integrity constraints *ric*.
- $OPA'$ associates to *op*′ an SQL statement whose first item is the second item of SQL statement of $OPA$, and whose second item is the first item of SQL statement of $OPA$.

In our running example, the object property associations for `module-diploma` and `diploma-module` object properties are:

```
ex:module-diploma ⇝
            SELECT Module.moduleId AS DOM, Diploma.diplomaId AS RNG
            FROM Module, Diploma
            WHERE Module.diplomaId = Diploma.diplomaId
ex:diploma-module ⇝
            SELECT Diploma.diplomaId AS DOM, Module.moduleId AS RNG
            FROM Diploma, Module
            WHERE Module.diplomaId = Diploma.diplomaId
```

**Step 5**

In this step, referential integrity constraints of tables that are in **Case 2** are transformed into object properties. The mappings of these object properties are similar to those in previous step.

In our running example, the object property associations for `student-diploma` and `diploma-student` object properties are:

```
ex:student-diploma ⇝
            SELECT Student.studentID AS DOM, Diploma.diplomaId AS RNG
            FROM Student, Diploma
            WHERE Student.diplomaId = Diploma.diplomaId
```

```
ex:diploma-student ⇝
            SELECT Diploma.diplomaId AS DOM, Student.studentID AS RNG
            FROM Diploma, Student
            WHERE Student.diplomaId = Diploma.diplomaId
```

**Step 6**

In this step, any non-key column is transformed into a datatype property. For a table $T$ that has a non-key column *col*, let us consider that $C$ is the class corresponding to $T$, and *dp* is the datatype property created from *col*. The mapping between *dp* and *col* is expressed using a datatype property association $DPA$ that associates *dp* with an SQL statement that has two items in SELECT clause. The first item is the identifier of $T$ table (its primary key), given an alias "`DOM`". The second item is the column *col* given an alias "`RNG`".

In our running example, the datatype property association for the datatype property `student-studentNumber` is:

```
ex:student-studentNumber ⇝
            SELECT Student.studentID AS DOM, Student.studentNumber AS RNG
            FROM Student
```

Now we have seen how to generate mappings using *Associations with SQL statements* specification. In the next subsection we will explain how to generate mappings using *DOML* specification.

#### 7.3.5.2 Mapping Generation using DOML

In this section, we will see how the generated mapping document is expressed using DOML language.

We will use the following prefixes:

- `"ex"` represents the namespace of the generated ontology,
- `"map"` represents the namespace of the mapping document, and
- `"doml"` represents the namespace of DOML language.

As we have seen in Section 6.3, a DOML mapping document includes: 1) a full description of the database schema, and 2) a set of mapping bridges including: concept bridges, datatype property bridges, and object property bridges.

Firstly, a description of the relational schema is automatically generated from the information of the database metadata. Each table is represented as an instance of `doml:Table`, it has `map:[tablename]-table` as its QName, where `[tablename]` is the table name. Each column is represented as an instance of `doml:Column`, it has `map:[tableName]-[columnName]-col` as its QName, where `[columnName]` and `[tablename]` are the names of the column and its owner table, respectively.

Figure 7.10 shows an excerpt of the DOML description of the database schema of our running example. This excerpt mentions the description of the table *Student* and its columns.

**Step 1**

In this step, an OWL class is generated from each database table in Case 3. A mapping between a class $C$ and its original table $T$ is expressed using a concept bridge whose `doml:class` is $C$, and whose `doml:toTable` is the RDF resource representing the table $T$ (an instance of `doml:Table`).

```
map:student-table    a                doml:Table;
                     doml:hasColumn  map:student-studentId-col,
                                     map:student-studentNumber-col,
                                     map:student-diplomaID-col;
                     doml:name       "student".
map:student-studentId-col         a        doml:Column;
                     doml:belongsToTable   map:student-table;
                     doml:columnType       xsd:integer;
                     doml:isPrimary        "true";
                     doml:name             "studentId";
                     doml:references       map:person-personId-col.
map:student-studentNumber-col      a       doml:Column;
                     doml:belongsToTable   map:student-table;
                     doml:columnType       xsd:string;
                     doml:name             "studentNumber".
map:student-diplomaID-col          a       doml:Column;
                     doml:belongsToTable   map:student-table;
                     doml:columnType       xsd:integer;
                     doml:name             "diplomaID";
                     doml:references       map:diploma-diplomaId-col.
```

FIGURE 7.10: An excerpt of the DOML description of the database schema

In our running example, the concept bridge for the concept `ex:Person` is:

```
map:person-cb     a             doml:ConceptBridge;
                  doml:class    ex:Person;
                  doml:toTable  map:person-table.
```

**Step 2**

In this step, an OWL class $C$ is generated from database table $T$ in Case 2, and it is set as subclass of the class $C_1$ corresponding to the referenced table $T_1$. A mapping between the class $C$ and the table $T$ is expressed as a concept bridge whose `doml:class` is $C$, and whose `doml:toTable` is the RDF resource the table representing $T$ (an instance of`doml:table`).

In our running example, the concept bridge for `ex:Student` class is:

```
map:student-cb   a             doml:ConceptBridge;
                 doml:class    ex:Student;
                 doml:toTable  map:student-table.
```

However, the properties that $C$ inherits from $C_1$ will have their own property bridges that are slightly different from property bridges of $C_1$ properties, as we will see in Step 6.

**Step 3**

In this step, there is a table $T$ which is in **Case 1** and has two referential integrity constraints: $ric_1(T, A_1, T_1, P(T_1))$ and $ric_2(T, A_2, T_2, P(T_2))$. Let us consider that $C_1$ and $C_2$ are the two classes corresponding to $T_1$ and $T_2$ respectively. Let us also consider that $cb_1$ and $cb_2$ are the two concept bridges that relate $C_1$ with $T_1$ and $C_2$ with $T_2$ respectively: $cb_1 = CB(C_1, T_1)$, $cb_2 = CB(C_2, T_2)$.

In this step, no OWL class is generated, but we assign to $C_1$ an object property $op_1$ whose range is $C_2$, and assign to $C_2$ an object property $op_2$ whose range is $C_1$. These two object properties ($op_1$ and $op_2$) are inverse. For these two object properties, we use two object property bridges $OPB_1$ and $OPB_2$:

- $OPB_1$, for the object property $op_1$, belongs to $cb_1$ and refers to $cb_2$. That is, its domain-concept-bridge (DCB) is $cb_1$, and its range-concept-bridge (RCB) is $cb_2$.
- $OPB_2$, for the object property $op_2$, belongs to $cb_2$ and refers to $cb_1$. That is, its domain-concept-bridge (DCB) is $cb_2$, and its range-concept-bridge (RCB) is $cb_1$.

Both of these object property bridges have two join expressions according to the referential integrity constraints $ric_1$ and $ric_2$ respectively. That is, each one of these bridges have two `doml:join-via` indicating the expressions: $T.pfk_1 = T_1.pk$ and $T.pfk_2 = T_2.pk$, where $A_1 = \{pfk_1\}$, $A_2 = \{pfk_2\}$, $T_1.pk \in P(T_1)$ , and $T_2.pk \in P(T_2)$.

These two bridges can be simply noted as:

$$OPB_1 = OPB(op_1, cb_1, cb_2, JOIN(T.pfk_1 = T_1.pk \text{ AND } T.pfk_2 = T_2.pk))$$
$$OPB_2 = OPB(op_2, cb_2, cb_1, JOIN(T.pfk_1 = T_1.pk \text{ AND } T.pfk_2 = T_2.pk))$$

In our running example, the object property bridgess for `ex:presence-session` and `ex:presence-student` object properties are:

```
map:presence-session-OPB       a                          doml:ObjectPropertyBridge;
                  doml:objectProperty             ex:presence-session ;
                  doml:belongsToConceptBridge     map:student-cb;
                  doml:refersToConceptBridge      map:session-cb;
                  doml:join-via                   map:join1.
map:presence-student-OPB       a                          doml:ObjectPropertyBridge;
                  doml:objectProperty             ex:presence-student;
                  doml:belongsToConceptBridge     map:session-cb;
                  doml:refersToConceptBridge      map:student-cb;
                  doml:join-via                   map:join1.
map:join1         a                               doml:Condition;
                  doml:conditionType              doml:AND;
                  doml:hasArguments               map:arg-list-join.
map:arg-list-join a                               doml:ArgList;
                  rdf:_1                          map:cond1;
                  rdf:_2                          map:cond2.
map:cond1         a                               doml:Condition;
                  doml:conditionType              doml:Equals;
                  doml:hasArguments               map:arg-list1.
map:arg-list1     a                               doml:ArgList;
                  rdf:_1                          map:presence-studentId-col;
                  rdf:_2                          map:student-studentId-col.
map:cond2         a                               doml:Condition;
                  doml:conditionType              doml:Equals;
                  doml:hasArguments               map:arg-list2.
map:arg-list2     a                               doml:ArgList;
                  rdf:_1                          map:presence-sessionId-col;
                  rdf:_2                          map:session-sessionId-col.
```

**Step 4**

In this step, referential integrity constraints of tables that are in **Case 3** are transformed into object properties. Let us consider a table $T$ which is in **Case 3**, and has a referential integrity constraint $ric(T, A, T_1, A_1)$, and $C$ and $C_1$ are the classes corresponding to $T$ and $T_1$ respectively.

Let us also consider that *cb* and $cb_1$ are the concept bridges that relate $C$ with $T$, and $C_1$ with $T_1$ respectively: $cb = CB(C,T)$, $cb_1 = CB(C_1,T_1)$.

In this step, we assign to $C$ an object property *op* whose range is $C_1$, and we assign to $C_1$ an object property $op'$ (inverse of *op*) whose range is $C$. For these two object properties *op* and $op'$, we use two object property bridges $OPB$ and $OPB'$:

- $OPB$ concerns the object property *op*, belongs to *cb* and refers to $cb_1$.
- $OPB'$ concerns for the object property $op'$, belongs to $cb_1$ and refers to *cb*.

Both of these object property bridges have a join expression according to the referential integrity constraint *ric*: $T.fk_1 = T_1.pk$. These two bridges can be simply noted as:

$OPB = OPB(op,\ cb,\ cb',\ JOIN(T.fk_1 = T_1.pk))$
$OPB' = OPB(op',\ cb',\ cb,\ JOIN(T.fk_1 = T_1.pk))$

In our running example, the object property bridges for `ex:module-diploma` and `ex:diploma-module` object properties are simply noted:

*module-diploma-OPB* = *OPB*(**ex:module-diploma**, *module-cb*, *diploma-cb*,
*JOIN*(*Module.diplomaId*=*Diploma.diplomaId*))
*diploma-module-OPB* = *OPB*(**ex:diploma-module**, *diploma-cb*, *module-cb*,
*JOIN*(*Module.diplomaId*=*Diploma.diplomaId*))

These bridges are encoded in DOML as follows:

```
map:module-diploma-OPB      a                          doml:ObjectPropertyBridge;
              doml:objectProperty          ex:module-diploma;
              doml:belongsToConceptBridge  map:module-cb;
              doml:refersToConceptBridge   map:diploma-cb;
              doml:join-via                map:join.
map:diploma-module-OPB      a                          doml:ObjectPropertyBridge;
              doml:objectProperty          ex:diploma-module;
              doml:belongsToConceptBridge  map:diploma-cb;
              doml:refersToConceptBridge   map:module-cb;
              doml:join-via                map:join2.
map:join2     a                            doml:Condition;
              doml:conditionType           doml:Equals;
              doml:hasArguments            map:arg-list2.
map:arg-list2 a                            doml:ArgList;
              rdf:_1                       map:module-diplomaId-col;
              rdf:_2                       map:diploma-diplomaId-col.
```

**Step 5**

In this step, referential integrity constraints of tables that are in Case 2 are transformed into object properties. The mappings of these object properties are similar to those in the previous step.

In our running example, the object property bridges for `ex:student-diploma` and `ex:diploma-student` object properties are:

*student-diploma-OPB* = *OPB*(**student-diploma**, *student-cb*, *diploma-cb*,
JOIN(*Student.diplomaId*=*Diploma.diplomaId*))
*diploma-student-OPB* = *OPB*(**diploma-student**, *diploma-cb*, *student-cb*,
JOIN(*Student.diplomaId*=*Diploma.diplomaId*))

**Step 6**

In this step, any non-key column is transformed into a datatype property. For a table $T$ that has a non-key column *col*, let us consider that $C$ is the class corresponding to $T$, and *dp* is the datatype property created from *col*. Let us also consider that $cb = CB(C, T)$ is the concept bridges that relate $C$ with $T$.

The mapping between *dp* and *col* is expressed using a datatype property bridge $DPB$ that belongs to *cb* and whose `doml:toColumn` refers to the RDF resource representing the column *col* (an instance of `doml:Column`).

In our running example, the datatype property association for the datatype property `ex:student-studentNumbe`
is:

```
map:student-studentNumber-DPB          a           doml:DatatypePropertyBridge;
               doml:datatypeProperty               ex:student-studentNumber;
               doml:belongsToConceptBridge         map:student-cb;
               doml:toColumn                       map:student-studentNumber-col.
```

**Inherited Properties**

In DOML, the properties that a (sub-)concept inherits from its ancestors will have their own property bridges that are slightly different from original property bridges (bridges of the ancestor properties).

Firstly, the bridge of the inherited property should belong to the domain-concept-bridge of the sub-concept (instead of the bridge of the ancestor concept).

In addition, a join expression is needed to relate the tables corresponding to the sub- and super-concepts.

For example, the concept `ex:Student` inherits the datatype property `ex:person-firstName` from its super-concept `ex:Person`. Thus, a datatype property bridge is specifically defined for the property `ex:person-firstName` inherited by `ex:Student`. This bridge belongs to the concept bridge `map:student-cb` of the concept `ex:Student`, and has a condition that indicates how the tables *Student* and *Person* (mapped to the concept `ex:Student` and `ex:Person` respectively) are joined: *student.studentId* = *person.personId*

Figure 7.11 shows the datatype property bridges for the general datatype property (whose domain is `ex:Person`) and the special datatype property (whose domain is `ex:Student`):

## 7.4 Mapping a Database to an Existing Ontology

The second goal of DB2OWL tool is to map a relational database to an existing ontology. DB2OWL allows a human expert to manually indicate the mappings between the components of a database and an existing ontology. This mapping process does not require to change the ontology nor the database. The result of this process is a mapping document that describes the correspondences between the database and the ontology. This mapping document can be

```
map:person-firstName-DPB    a           doml:DatatypePropertyBridge;
          doml:datatypeProperty         ex:person-firstName;
          doml:belongsToConceptBridge   map:person-cb;
          doml:toColumn                 map:person-firstName-col.

map:student-firstName-DPB   a           doml:DatatypePropertyBridge;
          doml:datatypeProperty         ex:person-firstName;
          doml:belongsToConceptBridge   map:student-cb;
          doml:toColumn                 map:person-firstName-col;
          doml:when                     map:student-person-join.
map:student-person-join     a           doml:Condition;
          doml:conditionType            doml:Equals;
          doml:hasArguments             map:argList1.
map:argList1   a          doml:ArgList;
               rdf:_1     map:student-studentId-col;
               rdf:_2     map:person-personId-col.
```

FIGURE 7.11: Datatype property bridges for the property `ex:person-firstName`

expressed using one of the two proposed kinds of mapping specifications: Associations with SQL statements or DOML.

However, the current implementation of DB2OWL provides a graphical interface that allows the user to provide mappings in the form of "Associations with SQL statements". In the future, we will develop a version of DB2OWL that allows to provide mappings specified using DOML language.

### 7.4.1 Associations with SQL statements

Mapping is done by selecting from the database an SQL statement and associating it to an ontology class or property. The new association is added to the mapping project. The global mapping process consists of the following steps:

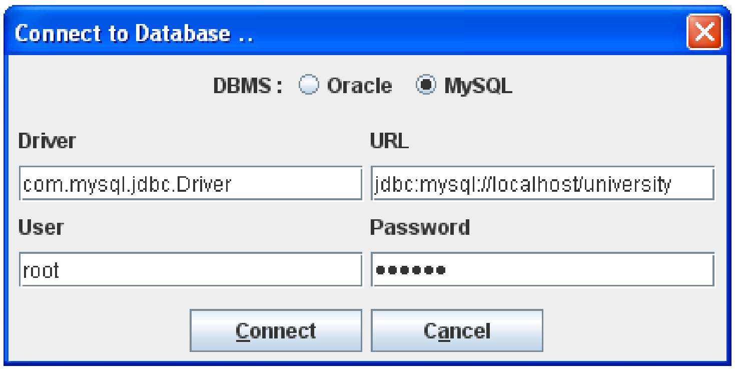
1. Connect to the database.
2. Select ontology file.
3. Map ontology components to database components.
    (a) Select a component (a class or a property) from the ontology.
    (b) Formulate an SQL statement to associate with the selected component.
    (c) Add the formed association to the mapping project.
4. Save the mapping project to a mapping document in XML format.

The final result is a mapping document that includes associations of ontology components with SQL statements that return corresponding datasets.

#### 1. Connect to the database

As shown in Figure 7.12, DB2OWL offers a dialog box that allows the user to provide necessary information needed to connect to the database. Using this dialog box, the user can indicate:

1. The DBMS hosting the database. Currently, DB2OWL supports MySQL and Oracle DBMSs.
2. The JDBC driver class name for the database. For example, the driver name for MySQL database is: com.mysql.jdbc.Driver.
3. The JDBC database URL. This is a string of the form `jdbc:subprotocol:subname`. For example, the database URL for a MySQL database is something like `jdbc:mysql://hostname:port/dbname`.
4. The username and the password required by the database.


Connect to Database ..

DBMS : Oracle [✓ MySQL]

Driver: com.mysql.jdbc.Driver

URL: jdbc:mysql://localhost/university

User: root

Password: ••••••

FIGURE 7.12: DB2OWL - connect to the database

When DB2OWL successfully connects to the database, the user can browse the database tables and their columns (see Figure 7.13). For each column, the user can see the name, datatype, whether it is a primary key. If the column is a foreign key, the user can also see the reference table and reference column.

## 2. Select the ontology file

DB2OWL allows the used to choose the file containing the OWL ontology. When the ontology is loaded, the user can navigate the hierarchy of ontology classes (Figure 7.14). For each OWL class, the user can see its super classes, as well as, its properties.

The user can also navigate the list of ontology properties, that also shows their domain and range (Figure 7.15).

## 3. Map ontology components to database components

Now, as the database is connected and the ontology is loaded, the user can start establishing mapping associations. For each mapping association, the user firstly select a component from the ontology (to select a class, double click on the class node in the hierarchy, to select a property, click on "Edit Mapping" button).

When an ontology component is selected, a window titled "Edit SQL" appears (see Figure 7.16). This window allows the user to formulate an SQL statement that he/she desires to associate with the selected ontology component. It offers several facilities that assist the user to build this SQL statement, such as 1) drag and drop editing facility, 2) automatic verification of the syntax of SQL statement, and 3) automatic insertion of join statements based on foreign keys.

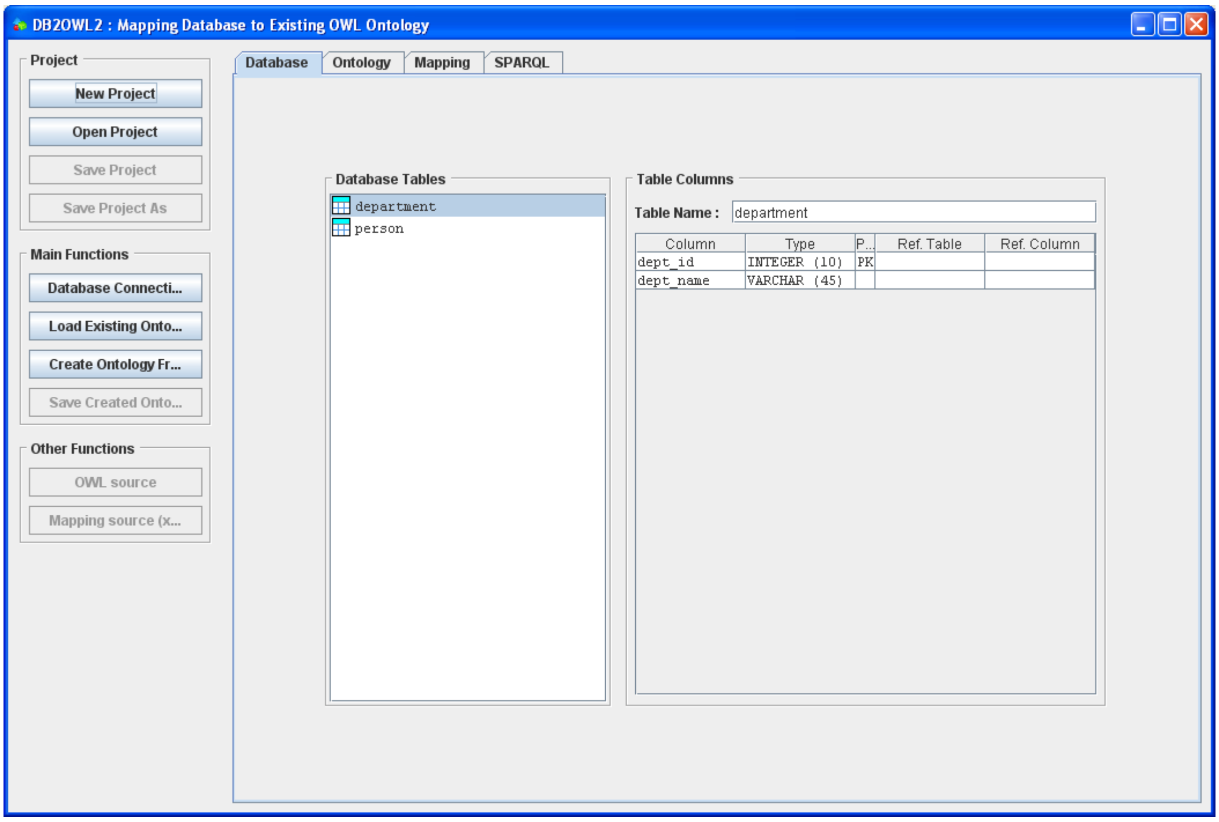


Table Name : department

| Column | Type | P... | Ref. Table | Ref. Column |
|---|---|---|---|---|
| dept_id | INTEGER (10) | PK | | |
| dept_name | VARCHAR (45) | | | |

FIGURE 7.13: DB2OWL - database schema browser

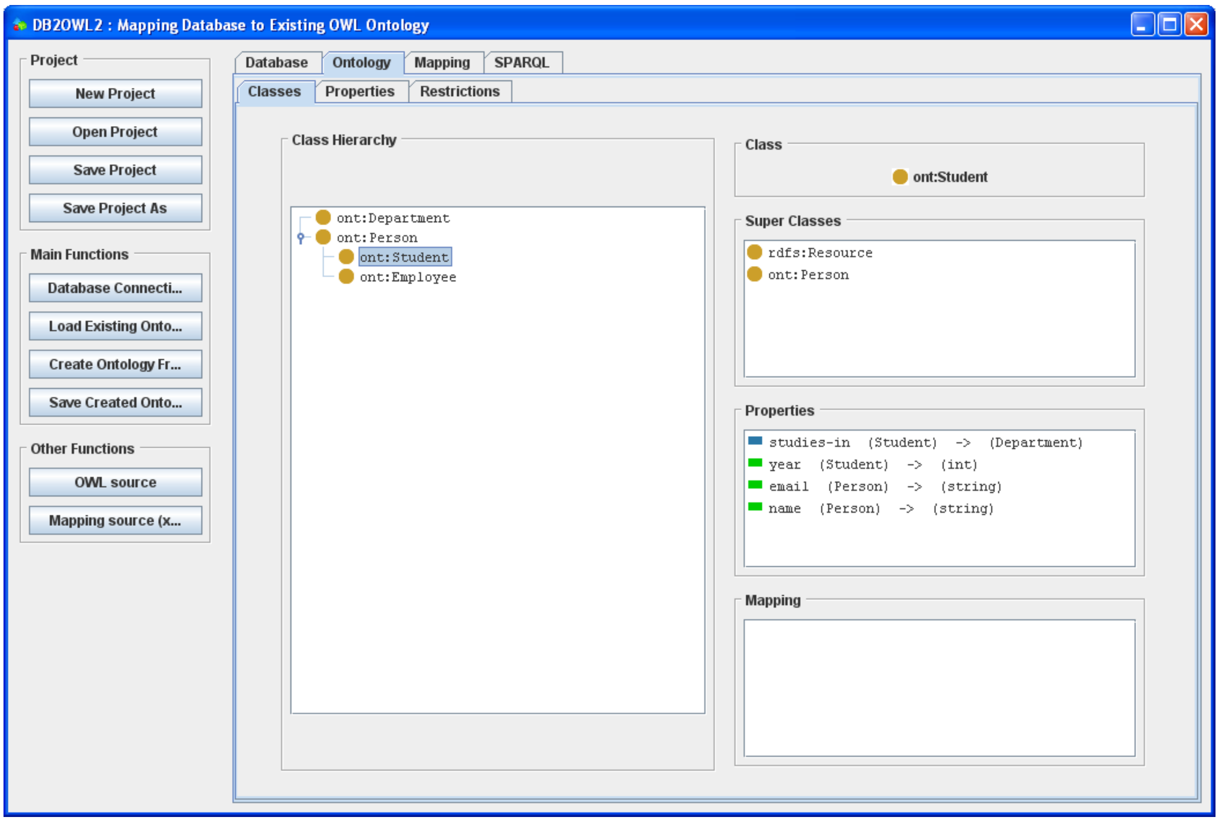
The "Edit SQL" window consists of (see Figure 7.16):

1. a list of database tables and columns (left side),
2. some text fields for editing SQL clauses: SELECT, FROM, and WHERE (right-top side),
3. a text area showing the result SQL statement (right-bottom side).

Remember that in "*Association with SQL statements*" mapping specification, the SQL statement associated with an ontology concept has exactly one item in the SELECT clause (aliased "`DOM`"), whereas the SQL statement associated with an ontology property has exactly two items in the SELECT clause (aliased "`DOM`" and "`RNG`" respectively). Therefore, the right-top side of the window has two text fields for the two possible items of the SELECT clause, a text field for the FROM clause, and a text area for the WHERE clause.

The user can build the SQL statement by either: 1) writing directly inside appropriate text fields, or 2) dragging database tables and columns from the list and dropping them into appropriate text fields. Using the drag and drop facility, several points are taken into account:

1. Column items are not allowed to be dropped into the text field of the FROM clause.
2. Table items are not allowed to be dropped into the text fields of the SELECT clause.
3. When the user drag and drop a column into the text fields of the SELECT clause, the table of this column is automatically added to the text field of the FROM clause.
4. When the user drag and drop a column which is a foreign key, then a suitable join statement (based on this foreign key) is automatically inserted into the text area of the WHERE clause.



```
ont:Department
ont:Person
    ont:Student
    ont:Employee
```

Class: ont:Student

Super Classes:

```
rdfs:Resource
ont:Person
```

Properties:

```
studies-in  (Student)  ->  (Department)
year  (Student)  ->  (int)
email  (Person)  ->  (string)
name  (Person)  ->  (string)
```

FIGURE 7.14: DB2OWL - hierarchy of ontology classes

Throughout this editing phase, the result SQL statement (shown in the text area of the right-bottom side of the window) changes dynamically according to the changes in the text fields of the right-top side, and it is checked to verify whether it is syntactically valid or not.

When the user is satisfied with the final SQL statement, he/she can approve the changes, thus, the SQL statement is associated with selected ontology component, and the mapping association is added to the mapping project. Only syntactically valid SQL statements are accepted. If the user approve an invalid SQL statement, the association will be rejected and not added to the mapping project.

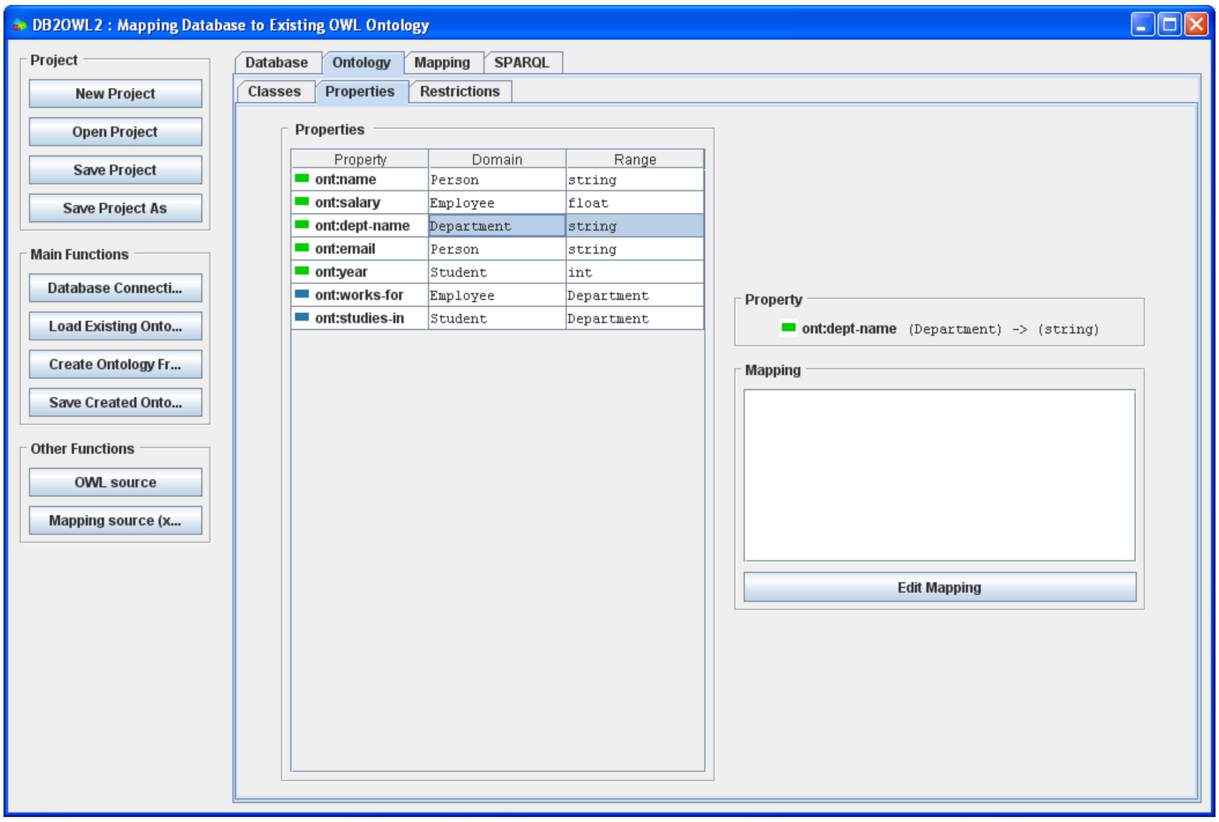
### 4. Save the mapping project to a mapping document in XML format

The established mapping associations are added to the mapping project. DB2OWL depicts mapping associations in two tables, one for class associations (Figure 7.17) and the second for property associations (Figure 7.18).

In these tables, each row includes an ontology component (class or property), its associated SQL statement, and a small button for re-editing the mapping association.

At the end of the mapping process, the user can save the mapping project to an XML file having the format explained in Section 6.2. This XML file represents the mapping document that will be used later during the query translation process.

In the next section, we will discuss the implementation of DB2OWL tool.



| Property | Domain | Range |
|---|---|---|
| 🟩 ont:name | Person | string |
| 🟩 ont:salary | Employee | float |
| 🟩 ont:dept-name | Department | string |
| 🟩 ont:email | Person | string |
| 🟩 ont:year | Student | int |
| 🟦 ont:works-for | Employee | Department |
| 🟦 ont:studies-in | Student | Department |

🟩 ont:dept-name (Department) -> (string)

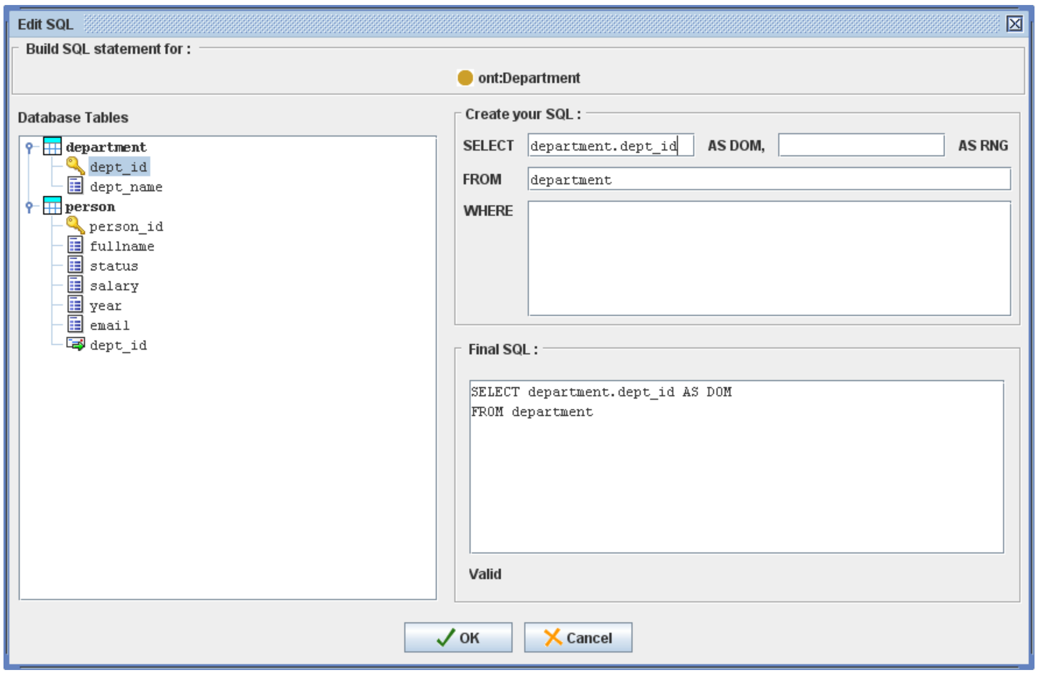
FIGURE 7.15: DB2OWL - list of ontology properties

## 7.5 Implementation

We have implemented a prototype of DB2OWL tool using Java programming language. This implementation is based on several freely-available APIs for java, namely: JDBC, Jena, Zql, and JDOM.

- **JDBC (Java Database Connectivity) API** [139] – using this API, we extract meta-data information about the database, including information about tables, columns, and constraints. To obtain these information we use java.sql.DatabaseMetaData interface[1] provided by JDBC, as well as some specific views provided by RDBMSs for describing the database metadata. More precisely, from Oracle, we use `USER_TABLES`, `USER_CONSTRAINTS`, and `USER_CONS_COLUMNS` views, and from MySQL, we use `TABLES`, `TABLE_CONSTRAINTS`, and `KEY_COLUMN_USAGE` views of the `INFORMATION_SCHEMA` catalog. The extracted metadata information are encapsulated into an internal database model.
- **Jena (Semantic Web Framework for Java)** [90] – Jena is a Java framework for building Semantic Web applications. It provides a programmatic environment for RDF, RDFS and OWL, SPARQL. We use this API for reading, manipulating and writting OWL ontologies.
- **Zql[2] (Java SQL parser)** – Zql parses SQL and fills in java structures representing SQL statements and expressions. We use this API to manipulate SQL statements of mappings.

[1]http://java.sun.com/j2se/1.4.2/docs/api/java/sql/DatabaseMetaData.html
[2]http://www.gibello.com/code/zql/



SELECT: department.dept_id

FROM: department

```
SELECT department.dept_id AS DOM
FROM department
```

FIGURE 7.16: DB2OWL - "Edit SQL" window

- **JDOM[3] (Java-based document object model for XML)** – JDOM provides a way to represent an XML document for easy and efficient reading, manipulation, and writing. We using this API to write the mapping project into XML format.

In the following subsection, we present some details on the implementation of both processes: ontology generation from the database, and mapping the database to an existing ontology.

### 7.5.1 Generating an Ontology from a Database

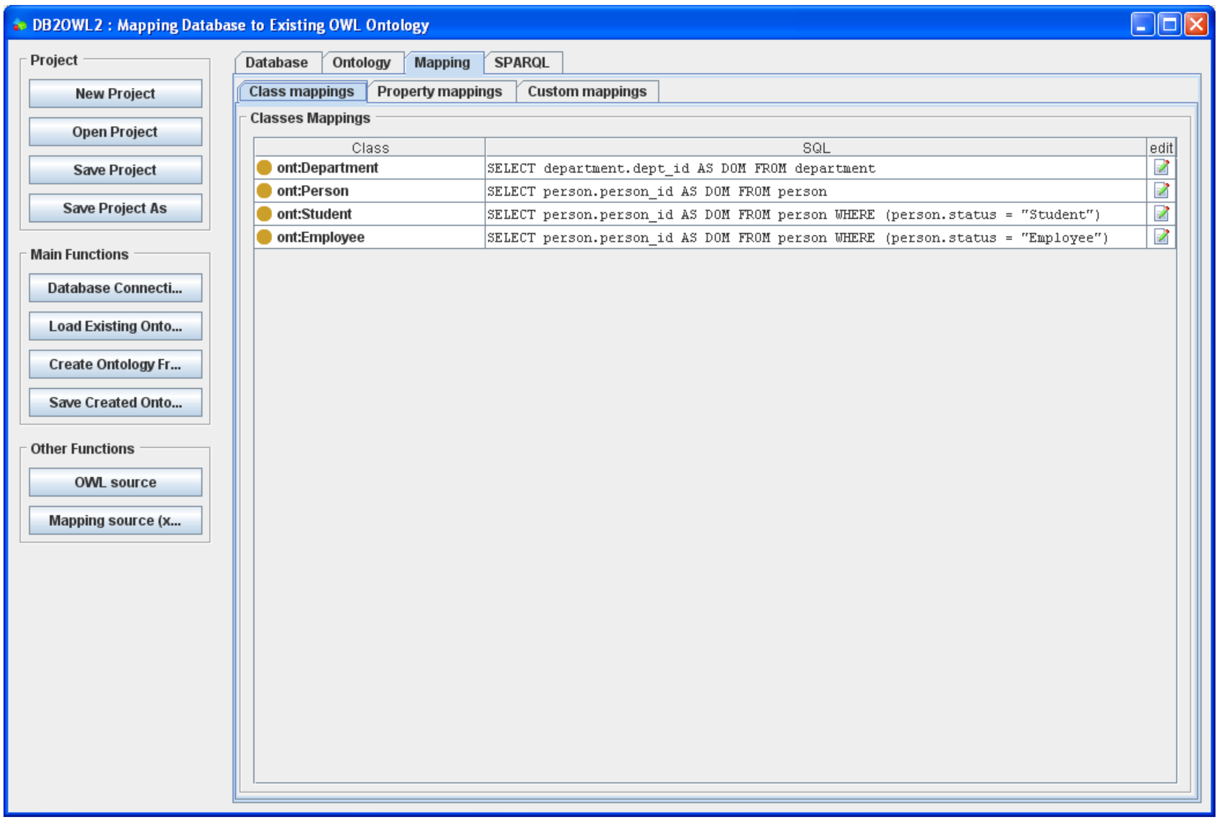
Firstly, metadata information about the database are extracted using JDBC API. These information are encapsulated as an internal database model. This database model is used as input to the ontology generation algorithm (see Section 7.3.5). Then, the execution of this algorithm generates an internal ontology model, which is transformed into Jena ontology model to obtain the final OWL ontology (see Figure 7.19).

During the execution of the algorithm, a mapping model is also generated that contains the associations between the components of the internal ontology model and the components of the internal database model. The Zql API is used to formulate SQL statements included in the associations. Finally, the mapping model is translated using JDOM API into XML format, yielding the mapping document.

### 7.5.2 Mapping a Database to an Exiting Ontology

Firstly, database metadata are extracted using JDBC API. These information are encapsulated as an internal database model. The existing ontology is loaded and expressed as Jena ontology

[3]http://www.jdom.org/



| Class | SQL | edit |
|---|---|---|
| ● ont:Department | SELECT department.dept_id AS DOM FROM department | |
| ● ont:Person | SELECT person.person_id AS DOM FROM person | |
| ● ont:Student | SELECT person.person_id AS DOM FROM person WHERE (person.status = "Student") | |
| ● ont:Employee | SELECT person.person_id AS DOM FROM person WHERE (person.status = "Employee") | |

FIGURE 7.17: DB2OWL - class mappings

model. Both the database and ontology models are presented to the user using a graphical user interface (GUI) developed using Swing framework.

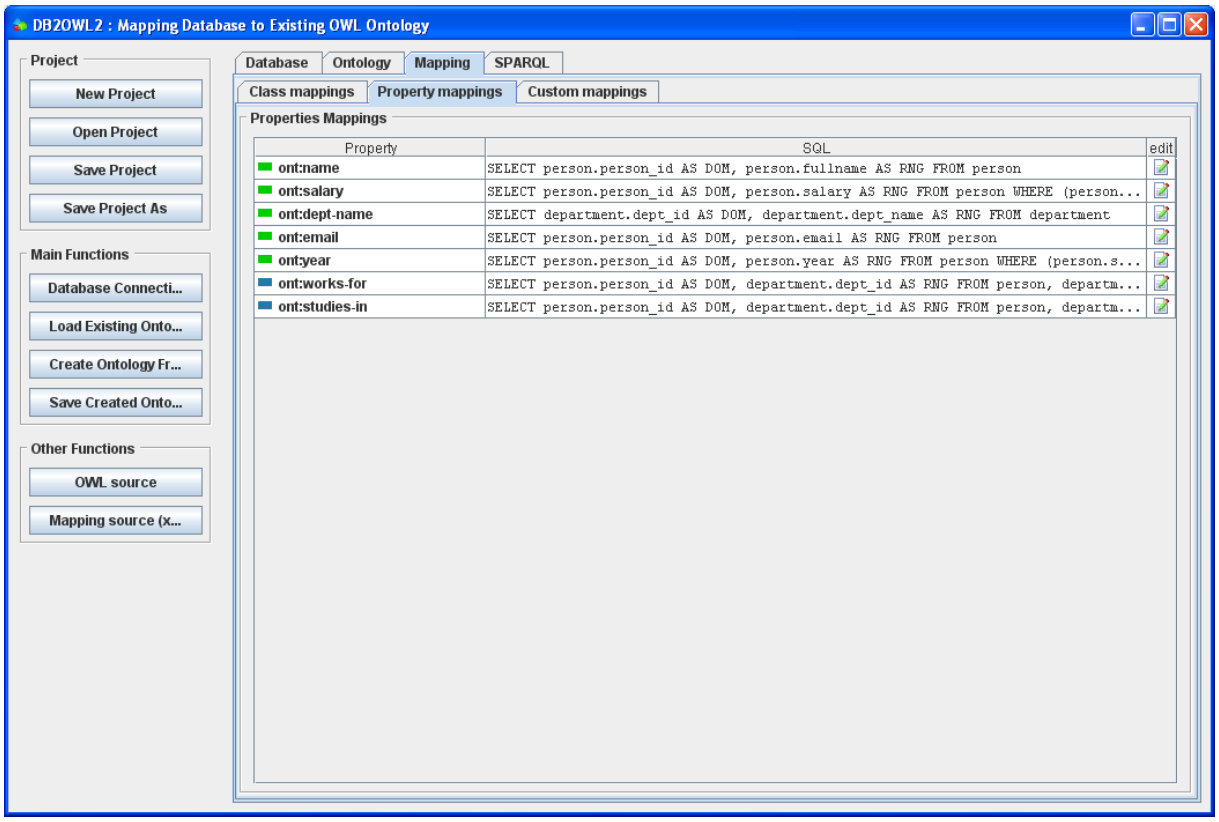
Using this GUI, the user can establish mapping associations between ontology components and SQL statements (based on database components). The Zql API is used to formulate SQL statements included in the associations, and to verify the validation of their syntax.

This database model is used as input to the ontology generation algorithm (see Section 7.3.4). Then, the execution of this algorithm generates an internal ontology model, which is transformed into Jena ontology model to obtain the final OWL ontology (see Figure 7.20). Finally, the mapping model is translated using JDOM API into XML format, yielding the mapping document.

Currently, DB2OWL supports Oracle and MySQL databases. We benefit from some specific views provided by these RDBMSs for describing the database metadata. DB2OWL can be easily extended to support other RDBMSs that provide such views.

## Chapter Summary

In this chapter, we have presented DB2OWL, our tool for database-to-ontology mapping. The purpose of this tool is twofold: 1) generate an ontology from a relational database, and 2) map a relational database to an existing OWL ontology. This tool is a part of the data provider module within OWSCIS architecture. It is intended to wrap local database to a the local ontology at the site level.



| Property | SQL | edit |
|---|---|---|
| 🟩 ont:name | SELECT person.person_id AS DOM, person.fullname AS RNG FROM person | |
| 🟩 ont:salary | SELECT person.person_id AS DOM, person.salary AS RNG FROM person WHERE (person... | |
| 🟩 ont:dept-name | SELECT department.dept_id AS DOM, department.dept_name AS RNG FROM department | |
| 🟩 ont:email | SELECT person.person_id AS DOM, person.email AS RNG FROM person | |
| 🟩 ont:year | SELECT person.person_id AS DOM, person.year AS RNG FROM person WHERE (person.s... | |
| 🟦 ont:works-for | SELECT person.person_id AS DOM, department.dept_id AS RNG FROM person, departm... | |
| 🟦 ont:studies-in | SELECT person.person_id AS DOM, department.dept_id AS RNG FROM person, departm... | |

FIGURE 7.18: DB2OWL - property mappings

In Section 7.2, we have given a general description of DB2OWL and its main tasks.

In Section 7.3, we have presented the first task of DB2OWL which is: *ontology generation from a database*. This task starts by detecting some particular cases for tables in the database schema (Section 7.3.3). According to these cases, each database component (table, column, constraint) is then converted to a corresponding ontology component (concept, property, etc.) (Section 7.3.4). During the ontology generation process, DB2OWL also generates a mapping document that describes the correspondences between the database and the generated local ontology (Section 7.3.5). This mapping document can be expressed using two kinds of mapping specifications: *Associations with SQL statements* and DOML.

In Section 7.4, we have presented the second task of DB2OWL which is: *mapping a database to an existing ontology*. DB2OWL provides a graphical interface that allows the user to provide mappings in two possible forms: Associations with SQL statements, and DOML. This manual mapping process is presented for the first type of mapping specification (Associations with SQL statements) in Section 7.4.1.

Finally, in Section 7.5, we have explained some details about the implementation of a prototype of DB2OWL.

In the next chapter, we will present the third task of DB2OWL tool, which is SPARQL-to-SQL query translation.

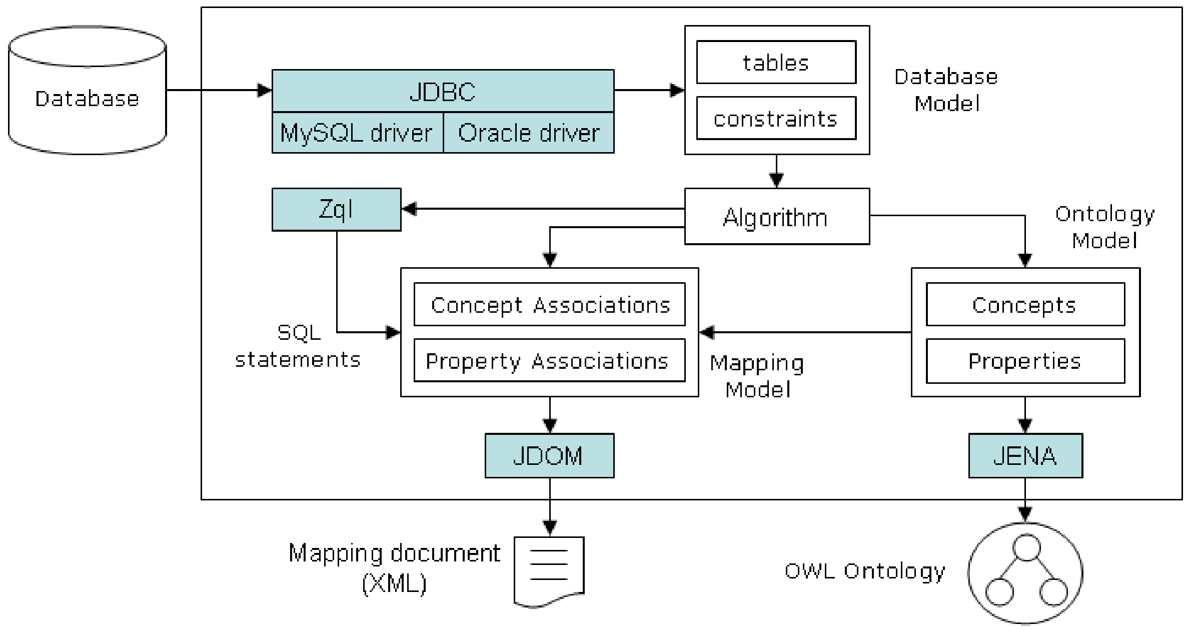


FIGURE 7.19: DB2OWL implementation: ontology generation from a database

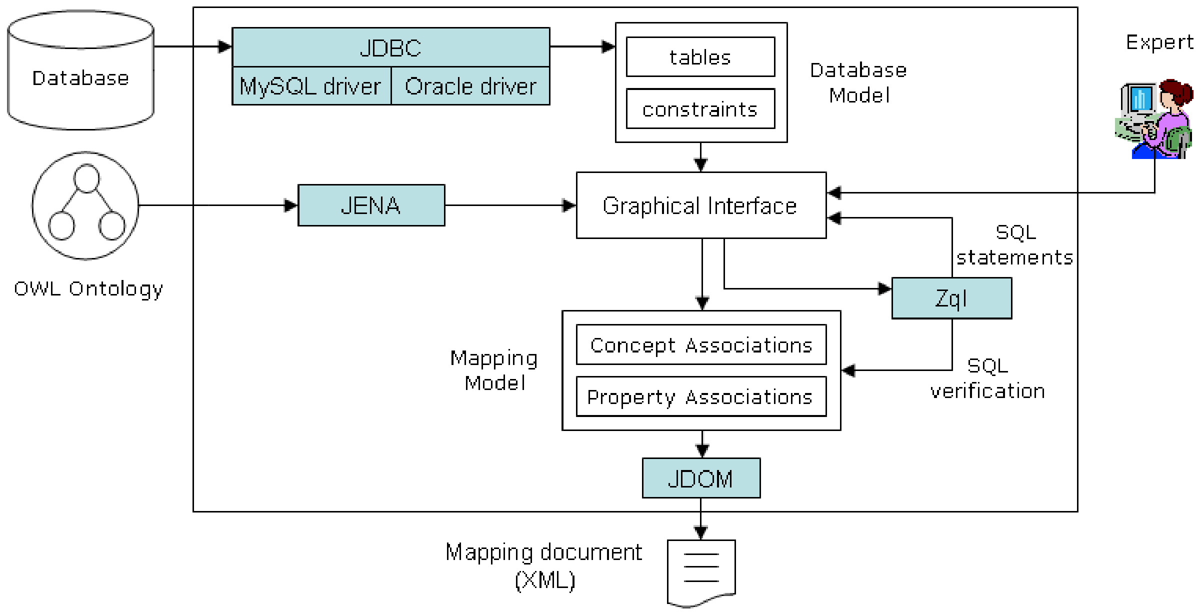


FIGURE 7.20: DB2OWL implementation: mapping a database to an exiting ontology

Chapter 8

# SPARQL-to-SQL Translation

Contents

## Abstract

In the previous chapter, we presented DB2OWL, our proposed tool for database-to-ontology mapping. We saw that this tool has three tasks:

1. create an ontology from a single database,
2. map a single database to an existing ontology, and
3. translate queries over an ontology towards queries over a single database.

The first two tasks were presented in the previous chapter.

In this chapter, we present the third task of DB2OWL which is the translation of SPARQL queries over an ontology into SQL queries over a mapped database.

We saw that DB2OWL supports two database-to-ontology mapping specifications: Associations with SQL Statements and DOML. For each of these two mapping specifications, we propose a suitable SPARQL-to-SQL translation method.

This chapter is organized as follows:

Firstly, in Section 8.1 we motivate the translation process. In Section 8.2 we introduce some preliminaries that we consider in the reminder on this chapter.

The first proposed translation method (for associations with SQL Statements) is presented in Section 8.3. This method consists of the following steps:

1. SPARQL query parsing (Section 8.3.1)
2. SPARQL-to-SQL query translation (Section 8.3.2)
3. Simplification of the translated SQL query (Section 8.3.3)

The second proposed translation method (for DOML mappings) is presented in Section 8.4. This method consists of the four steps:

1. Parsing of the DOML mapping document (Section 8.4.1).
2. SPARQL query parsing (Section 8.4.2).
3. Preprocessing step (Section 8.4.3).
4. SPARQL-to-SQL translation (Section 8.4.4).

In Section 8.5, we discuss the process of reformulation of SQL query results into SPARQL query results XML format. Finally, in Section 8.6, we talk about the implementation of proposed translation methods.

## 8.1 Motivation

OWSCIS is an ontology-based information integration system, within which local information sources are mapped to local ontologies that represent the semantics of those underlying sources. OWSCIS follows a non-materialized approach, that is data instances still reside in their sources, and are not materialized at ontology level. Ontology population is done using a query-driven mode, that is, ontology instances are only created to answer queries (only the data instances needed to answer a given query are transformed to ontology instances).

This approach requires a phase of query translation, where queries submitted over ontologies (using an ontology query language) are translated into corresponding queries over local information sources (using the native query languages of these sources, such as SQL for relational databases, and XQuery for XML data sources). We use SPARQL as the ontology query language in OWSCIS. An introductory presentation of this query language is given in Appendix F.

In Section 3.5, we gave an overview on query processing in OWSCIS system. We saw that the querying web service decomposes a user query into a set of sub-queries, and send them to the different data providers. Each data provider locally processes its sub-query and returns the results to the querying web service.

Figure 8.1 demonstrates the different steps needed to process a query within a data provider. When a sub-query is received by a data provider, it is firstly rewritten into a local-query in terms of the local ontology (in SPARQL). This rewriting phase is based on the mappings between the local ontology and the global ontology. The local (rewritten) query is then translated into a query over local information sources (in SQL for relational databases, and in XQuery for XML data sources). This translation phase is based on the mappings between the local ontology and underlying information sources. The translated query is then solved over the underlying data source, and their results are reformulated according to the local ontology. The reformulated results are returned to the querying web service.

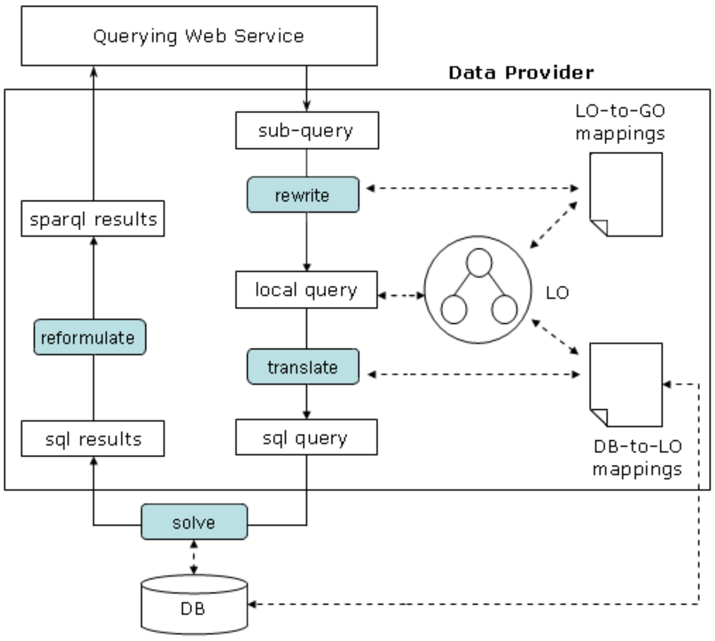


FIGURE 8.1: Query processing within a data provider

In this chapter, we will focus on the translation step. We will present this step in the case where the source is a single database. Thus, how to translate a local SPARQL query (in terms of the local ontology) into an SQL query in terms of the local relational database. Later, in Chapter 11, we will present the translation step in the case of XML data source, that is, SPARQL-to-XQuery translation.

In Chapter 7, we presented DB2OWL tool which is a part of the data provider module within OWSCIS architecture. It is intended to wrap local database to a local ontology at the site level. DB2OWL tool has three main tasks: 1) generate an ontology from a relational database, 2) map a relational database to an existing OWL ontology, and 3) translate SPARQL queries over an ontology into SQL queries over a mapped database. The first and the second tasks were presented in the previous chapter. In this chapter, we will present the third task of DB2OWL tool, which is SPARQL-to-SQL query translation.

In Appendix G, we make a survey on the existing works on SPARQL-to-SQL translation. We can observe, that these works are limited to the scope of RDF triple stores. To our knowledge, there are no works that handle SPARQL query translation into SQL over an arbitrary relational database.

In this chapter, we present our works about translating a SPARQL query into an equivalent SQL query over arbitrary relational database (as long as the required mappings are provided). Within OWSCIS framework, this translation is needed as a vital part in query processing inside data providers. A data provider receives queries in SPARQL language and rewrites them over its local ontology. These local SPARQL query need to be translated to the native language of the underlying information sources (SQL for relational databases). This translation phase is based on the mappings between the local ontology and underlying information sources.

However, in our translation algorithms that we present in the reminder of this chapter, we adopt some techniques from the presented existing works. For example, we use the BGP model of Chebotko et al [42] (see Section G.2).

We saw that DB2OWL supports two database-to-ontology mapping specifications: Associations with SQL Statements and DOML. For each of these two mapping specifications, we propose a suitable SPARQL-to-SQL translation method. In Section 8.3, we present our SPARQL-to-SQL translation method for the case where database-to-ontology mappings are specified using Associations with SQL statements. Later, in Section 8.4, we present our translation method for the case of mappings specified using DOML language. However, in the next section, we give some general preliminaries that we consider in both translation methods.

## 8.2 Preliminaries

The problem of SPARQL-to-SQL query translation that we attempt to solve in this chapter can be stated as follows:

We have an ontology $O$, and a relational database $D$. We submit a SPARQL query $Q_{SPARQL}$ over the ontology $O$, and we want to translate this query into an SQL query $Q_{SQL}$ over the database $D$.

Before presenting our translation methods, we will present some assumptions that we take in consideration. These assumptions concern: 1) Mapping covering, 2) Triple patterns, and 3) OPTIONAL clauses.

### 1) Mapping Covering

Firstly, we assume that a mapping document $M(D,O)$ is provided between the database $D$ and the ontology $O$. We consider that this mapping document covers the ontology. That is, for each ontology term $o \in O$, there is a mapping $m \in M(D,O)$ between this term and a corresponding database term (or expression) $d \in D$.

We consider two kinds of mapping specifications: Associations with SQL Statements (Section 6.2) and DOML language (Section 6.3). In the reminder of this chapter, we will present two translation methods for these two kinds of mapping specifications.

Section 8.3 presents the translation method in the case of mappings specified as Associations with SQL statements. In this case, a mapping $m$ is a (concept or property) association with SQL statement.

In Section 8.4, we present the translation method in the case of mappings specified in DOML language. In this case, a mapping $m$ is a (concept or property) mapping bridge.

### 2) OPTIONAL Clauses

We consider that the SPARQL query does not contain OPTIONAL clauses. That is, a SPARQL query is translatable using our method if it consists of a set of triple patterns and FILTER clauses only. The OPTIONAL clauses are not yet treated by our method (this would be a future work).

### 3) Triple Patterns

Finally, we set some conditions on acceptable triple patterns of the SPARQL query.

In SPARQL language, the subject can be a variable or a URI, the predicate can be a variable or a URI, and the object can be a variable, a URI or a literal. However, in our translation method:

- The subject should always be a variable that represents an instance of an ontology concept. URI subjects are not accepted, because the ontology does not contain matrialized instances. Thus, subjects (ontology instances) are unknown.
- The predicate should always be a URI. Variable predicates are not accepted, because they mean unknown database columns[1]. Thus, it is not possible to find an SQL query with unknown columns. More precisely, the predicate is either a (datatype or object) property in the ontology or `rdf:type`.
- The object can be a variable or a literal. In the case where the predicate is `rdf:type`, the object should be an ontology concept.

In summary, an acceptable triple pattern should have one of the forms shown in Table 8.1.

[1]rememebr, ontology properties correspond to database columns

| Subject | Predicate | Object | Example | | |
|---|---|---|---|---|---|
| `?var` | `[property]` | `[literal]` | `?x` | `:car-name` | `"Peugeot307"` |
| `?var1` | `[property]` | `?var2` | `?y` | `:works-for` | `?z` |
| `?var` | `rdf:type` | `[concept]` | `?y` | `rdf:type` | `:Person` |

TABLE 8.1: Acceptable triple patterns of SPARQL query

## 8.3 Translation using Associations with SQL Statements

In this section, we present a method for translating SPARQL queries into SQL queries in the case where mappings are provided in the form of Associations with SQL statements. This kind of mappings has been presented in Section 6.2.

### Running Example

In order to illustrate our translation methods, we will use the example given in Section 7.3, that was used in the previous chapter to illustrate the ontology generation process using DB2OWL tool. In this example, the database schema is shown in Figure 8.2, and the generated ontology is shown in Figure 7.9.

We will use the following SPARQL query (Figure 8.2). This query retrieves the names of modules given in the diploma named `"BDIA"`. The query is composed of one selected variable `?modName`, three triple patterns, and a filter that constrains the value of `?dipName` variable to be equals to `"BDIA"`.

```
SELECT  ?modName
WHERE {
    ?mod     ex:module-moduleName     ?modName.
    ?mod     ex:module-diploma        ?dip.
    ?dip     ex:diploma-diplomaName   ?dipName.
    FILTER(?dipName = "BDIA")
}
```

FIGURE 8.2: Example SPARQL Query 1

### The general Strategy

The vital step in the query translation is to establish a graph of the SPARQL query. This graph is called: Basic Graph Pattern (BGP) (adopted from [42], see Section G.2). A basic graph pattern is modeled as a directed graph $BGP = (N, E)$, where $N$ is a set of nodes representing subjects and objects, and $E$ is a set of edges representing predicates. Each edge is directed from a subject node to an object node. Each node is labeled with a variable name, a URI, or a literal, and each edge is labeled with a variable name or a URI. Additionally, each node has pointers to all incoming and outgoing edges and their quantities in the in-degree and out-degree attributes, respectively.

Then, each edge in this graph is associated with the suitable SQL statement that corresponds to the predicate node of the respective triple pattern. A unique alias is generated for this statement. The start node of the edge is associated to the first selected column in the statement (DOM) and the end node is associated to the second one (RNG).

The set of statements representing all the edges in the graph form the FROM clause of the final SQL query.

When two (or more) edges share a variable node, then their equivalent statements will be joined depending on the columns representing the shared node and the direction of the edge (incoming or outgoing). If an edge has a literal end node, then the equivalent statement will be restricted using a logical condition in which the column equivalent to the node equals the literal value.

The SELECT clause in SQL query will be the columns equivalent to variables in SELECT clause in SPARQL query. FILTER clauses are translated as conjunctive WHERE clauses. Logical and comparing operators in a FILTER clause are translated to equivalent operators in SQL and each variable mentioned in the FILTER is replaced by anyone of its equivalent columns from used statement.

The whole translation process consists of three main steps:

1. SPARQL query parsing,
2. SPARQL to SQL query translation, and
3. Optimization of the translated SQL Query.

In the following subsections, we explain these steps in details, and we use the running example for illustration.

### 8.3.1 SPARQL Query Parsing

In the first step, the SPARQL query is parsed and analyzed to get its different components, which are:

1. a set of selected variables $SV$,
2. a set of order variables $OV$,
3. a set of filters $FLTR$, and
4. a basic graph pattern $BGP = (N, E)$.

In Appendix H, Table H.1 shows the algorithm of query parsing. Firstly, the set of selected variables $SV$ is retrieved. If the SELECT clause contains an asterisk (“*”) that used to specify that the query should return all variables, then $SV$ includes all the variables mentioned in the query body. The set of order variables $OV$ is then retrieved.

The next thing to do is to retrieve the set of filters and establish the basic graph pattern. For each element of the query pattern elements, if the element is a filter element, then it is added to the filters set $FLTR$. If the element is a triple pattern, then we get its subject, predicate and object. For the subject and object, we create two vertices $v_{sub}$ and $v_{obj}$ and add them to the graph $BGP$. For the predicate, we create an edge eprd between the vertices $v_{sub}$ and $v_{obj}$. This edge is given a label (the predicate URI), and a unique alias. This edge is added to the graph $BGP$.

In our running example, the result of the parsing step is as follows:

```
SV={?modName}, OV={}
FLTR={(?dipName="BDIA")}
BGP=({?mod,?modName,?dip,?dipName},
    {(?mod,?modName) S1 ,(?mod,?dip) S2,(?dip,?dipName) S3})
```

Note that in the BGP, each edge has a label and an alias. The label corresponds to the predicate of the triple, which is either: `rdf:type` or an ontology property. The alias is unique for the scope of a SPARQL query. In our example, we gave the three edges, the following aliases: S1, S2, and S3.

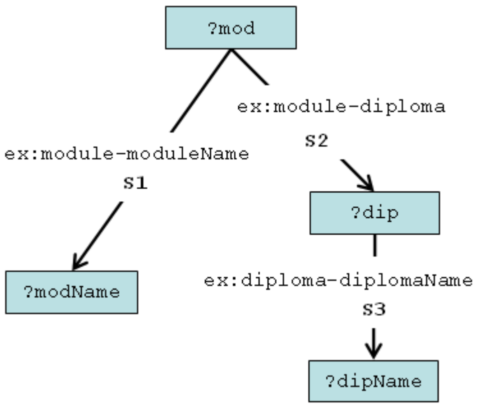


FIGURE 8.3: BGP of running example query

Now, the components of the SPARQL query are retrieved, they are passed as input to the next step which is *SPARQL-to-SQL Query Translation*.

### 8.3.2 SPARQL-to-SQL Query Translation

This is the second step of the whole query translation process. The input of this step is the output of the previous step plus a mapping document $MD$. A mapping document is a set of mapping associations $MD = \{assoc_j : j = 1, \cdots, m\}$, each of them corresponds to an ontology term.

Input:

1. a set of selected variables $SV$
2. a set of order variables $OV$
3. a set of filters $FLTR$
4. a basic graph pattern $BGP = (N, E)$
5. a mapping document $MD$

Output:

1. a set of SELECT items of the translated SQL query.
2. a set of FROM items of the translated SQL query.
3. a set of WHERE items of the translated SQL query.

In Appendix H, Table H.2 shows the algorithm of query translation. The whole process consists of the following steps:

1. Construct the FROM clause.
2. Construct the SELECT clause.
3. Construct the WHERE clause.
4. Build the SQL query.

#### 8.3.2.1 Construct the FROM Clause

The algorithm of this step is shown in Table H.2, lines 13-25. It goes as follows:

For each edge *e* in the graph BGP, we get its label *prd* and alias *al* (assigned through the previous step). The label (predicate) is an URI which is either `rdf:type` or an ontology property.

- If the predicate *prd* is `rdf:type`, then we get the target vertex of the edge *e*. It corresponds to the triple object which is an ontology concept. From the mapping document, we get the concept association of this ontology concept. We get the SQL statement *stmt* of this association. This statement is added to the set of FROM items, and given the respective alias *al*.
- Otherwise, the predicate *prd* is an ontology property. Then, from the mapping document, we get the mapping association *assoc* that corresponds to this property. We get the SQL statement *stmt* of this association. This statement is added to the set of FROM items, and given the respective alias *al*.

Hence, the FROM clause of the final SQL query is composed of bracketed SQL statements.

For example, the BGP of the query query1 contains three edges, they correspond to the properties: `ex:module-moduleName`, `ex:module-diploma`, and `ex:diploma-diplomaName`. They are given three aliases `S1`, `S2` and `S3`, respectively.

In the mapping document, these properties have three property associations that associate each of them with an SQL statement as shown in Table 8.2.

| **Ontology Term** | **SQL statement** |
|---|---|
| `ex:module-moduleName` | `SELECT Module.moduleId AS DOM, Module.moduleName AS RNG`<br>`FROM Module` |
| `ex:module-diploma` | `SELECT Module.moduleId AS DOM, Diploma.diplomaId AS RNG`<br>`FROM Module, Diploma`<br>`WHERE Module.diplomaId = Diploma.diplomaId` |
| `ex:diploma-diplomaName` | `SELECT Diploma.diplomaId AS DOM, Diploma.diplomaName AS RNG`<br>`FROM Diploma` |

TABLE 8.2: Mapping associations for the terms of example query1

The FROM clause of the final SQL query will be:

```
FROM (
     SELECT Module.moduleId AS DOM, Module.moduleName AS RNG
     FROM Module
  ) AS S1, (
     SELECT Module.moduleId AS DOM, Diploma.diplomaId AS RNG
     FROM Module, Diploma
     WHERE Module.diplomaId = Diploma.diplomaId
  ) AS S2, (
     SELECT Diploma.diplomaId AS DOM, Diploma.diplomaName AS RNG
     FROM Diploma
  ) AS S3
```

### 8.3.2.2 Construct the SELECT clause

The SELECT clause in the final SQL query will be the database columns that are equivalent to the selected variables *SV* of the original SPARQL query (variables in the SELECT clause).

Firstly, every variable in the SPARQL query will be coupled with one or more aliased columns. We mean by aliased column: an alias of an SQL statement (e.g. `"S1"`,`"S2"`, $\cdots$) plus an alias `".DOM"` or `".RNG"` (according to the direction of the incoming or outgoing edges of the node).

For example, Figure 8.4 shows the aliased columns coupled with the variable nodes of query1:

- `?mod` is coupled with: `S1.DOM` and `S2.DOM` (the start node of the edges aliased `S1` and `S2`).
- `?modName` is coupled with: `S1.RNG` (the end node of the edge aliased `S1`).
- `?dip` is coupled with: `S2.RNG` and `S3.DOM` (the start node of the edge aliased `S2` and the end node of the edge aliased `S3`).
- `?dipName` is coupled with: `S3.RNG` (the end node of the edge aliased `S3`).

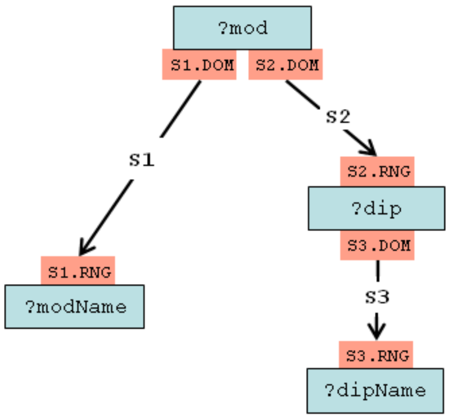


FIGURE 8.4: The aliased columns coupled with the variable nodes of query1

For each selected variable, we place one of its aliased columns into the SELECT clause of the SQL query and give it an alias which is the name of the variable.

For example, query1 has one selected variable `?modName`. This variable is coupled with one aliased column: `S1.RNG`. Thus, the SELECT clause of the final SQL query will be:

```
SELECT S1.RNG AS modName
```

The algorithm of this step is shown in Table H.2, lines 27-45. It goes as follows:

- For each node $n$ in the BGP, if the node is labeled with a variable $var$, then we get its first incoming edge $e_1^{in}$ (if any). We form an aliased column by concatenating the alias of this edge $e_1^{in}$ with `".RNG"` (because this edge comes in the node $n$, thus this node is the end/range of that edge). If the variable $var$ is selected, then the aliased column is given an alias the name of the variable $var$, and added to the SELECT clause of the SQL query.
- If the node has no incoming edges, we get its first outgoing edge $e_1^{out}$ (there must be at least one). We form an aliased column by concatenating the alias of this edge $e_1^{out}$ with `".DOM"` (because this edge goes out the node $n$, thus this node is the start/domain of that edge). If the variable $var$ is selected, then the aliased column is given an alias the name of the variable $var$, and added to the SELECT clause of the SQL query.

#### 8.3.2.3 Construct the WHERE clause

The WHERE clause of the final SQL query is constructed using three kinds of conditions:

- Join conditions resulting from shared variables.
- Conditions resulting from literal values restrictions.
- Conditions translated from FILTER clauses.

**Join conditions resulting from shared variables**

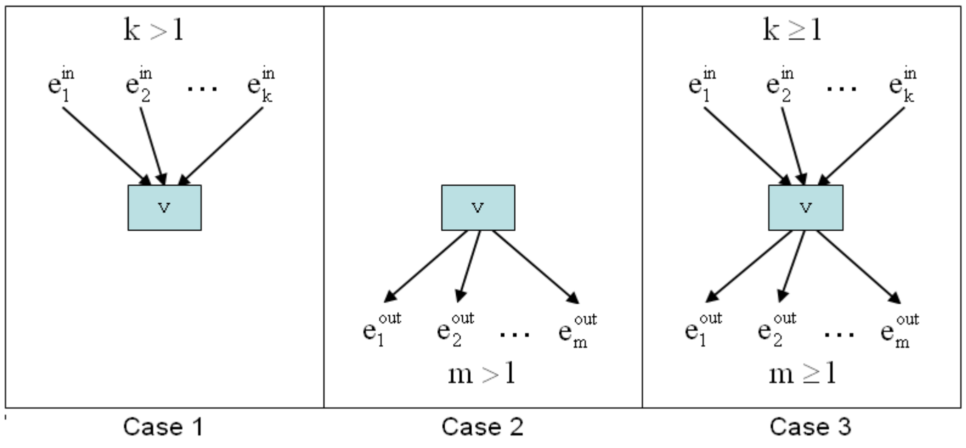
When two (or more) edges share a variable node, then their associated SQL statements will be joined using the columns (with aliases `"DOM"` and `"RNG"`) representing the shared node and depending on the direction of the edge (incoming or outgoing). We distinguish three cases:

- **Case 1** – A variable node $v$ has multiple incoming edges: $e_1^{in}, e_2^{in}, \cdots, e_k^{in} : k > 1$ (Figure 8.5 - Case 1). In this case, the `"RNG"` column of all SQL statements aliases that correspond to these incoming edges must be equal.
$$e_1^{in}.RNG = e_2^{in}.RNG$$
$$\dots$$
$$e_1^{in}.RNG = e_k^{in}.RNG$$
- **Case 2** – A variable node $v$ has multiple outgoing edges: $e_1^{out}, e_2^{out}, \cdots, e_m^{out} : m > 1$ (Figure 8.5 - Case 2). In this case, the `"DOM"` column of all SQL statements aliases that correspond to these outgoing edges must be equal.
$$e_1^{out}.DOM = e_2^{out}.DOM$$
$$\dots$$
$$e_1^{out}.DOM = e_m^{out}.DOM$$
- **Case 3** – (This is the most general case) A variable node $v$ has one or more incoming edges: $e_1^{in}, e_2^{in}, \cdots, e_k^{in} : k \geq 1$ and one or more outgoing edges: $e_1^{out}, e_2^{out}, \cdots, e_m^{out} : m \geq 1$ (Figure 8.5 - Case 3). In this case, we join incoming edges together as in Case 1, and we join outgoing edges together as in Case 2, and we join the first incoming edge with the first outgoing edge. That is, the `"RNG"` column of the SQL statement alias that correspond to the first incoming edge $e_1^{in}$ must be equal to the `"DOM"` column of the SQL statement alias that correspond to the first outgoing edge $e_1^{out}$.

$$e_1^{in}.RNG = e_1^{out}.DOM$$

**Example** – In our running example, in the BGP of query1 (shown in Figure 8.3), there are two shared variables: `?mod` and `?dip`.

- The variable node `?mod` is shared between two edges aliased with `S1` and `S2`. Both of these edges are outgoing of the variable node. Thus, according to Case 2, these edges are joined using the column `"DOM"` of both. The join condition is:
  `S1.DOM = S2.DOM`
- The variable node `?dip` is shared between two edges aliased with `S2` and `S3`. The first edge `S2` is incoming to the shared node `?dip`, while the second edge `S3` is outgoing of it. Thus, according to Case 3, the `"RNG"` column of the incoming edge `S2` must be equal to the `"DOM"` column of the outgoing edge `S3`. Thus, the join condition is:
  `S2.RNG = S3.DOM`



k > 1
$e_1^{in}$ $e_2^{in}$ … $e_k^{in}$
v
Case 1
v
$e_1^{out}$ $e_2^{out}$ … $e_m^{out}$
m > 1
Case 2
k ≥ 1
$e_1^{in}$ $e_2^{in}$ … $e_k^{in}$
v
$e_1^{out}$ $e_2^{out}$ … $e_m^{out}$
m ≥ 1
Case 3

FIGURE 8.5: Three cases of joins resulting from shared variables

### Conditions resulting from literal values restrictions

This case occurs for nodes labeled with literal values. When a literal node has an incoming edge $e_{in}$, then the `"RNG"` column of the SQL statement alias that correspond to this incoming edge must be equal to that literal value.

$$e^{in}.RNG = [value]$$

**Note** – literal nodes cannot have outgoing edges, because in SPARQL triple patterns, subjects cannot be literals.

### Conditions translated from FILTER clauses

FILTER clauses are translated to equivalent SQL conditions. This translation is straightforward:

- Literal values are not changed.
- Comparing operators (=, <, <=, $\cdots$) are not changed (they exist in SPARQL and SQL).
- SPARQL logical operators (&&, ||, !) are replaced by equivalent SQL logical operators (AND, OR, NOT).
- Each variable mentioned in the FILTER is replaced by someone of its coupled aliased columns (computed at the step of construct SELECT clause).

**Example** – In the example query 1, the filter expression: `(?dipName="BDIA")` is translated to the SQL condition `(S3.RNG="BDIA")`, since the variable `?dipName` was coupled with the aliased column `S3.RNG` (see section 8.3.2.2).

The whole WHERE clause of query1 consists of two join conditions and a FILTER translated condition:

```
WHERE  S1.DOM = S2.DOM
AND    S2.RNG = S3.DOM
AND    S3.RNG = "BDIA"
```

#### 8.3.2.4 Build the SQL query

To this point, we obtained the sets of SELECT items $\{\sigma_1, \sigma_2, \cdots, \sigma_k\}$, FROM items $\{\phi_1, \phi_2, \cdots, \phi_l\}$ and WHERE items $\{\omega_1, \omega_2, \cdots, \omega_m\}$ from the previous steps. We also have had the set of order items: $OV = \{\upsilon_1, \upsilon_2, \cdots \upsilon_n\}$ from the *SPARQL query Parsing* step. Thus, we can simply build the target SQL query $Q_{SQL}$ using the items of these different sets as follows:

```
SELECT $\sigma_1, \sigma_2, \cdots, \sigma_k$
FROM $\phi_1, \phi_2, \cdots, \phi_l$
WHERE $\omega_1$ AND $\omega_2$ AND $\cdots$ AND $\omega_m$
ORDER BY $\upsilon_1, \upsilon_2, \cdots, \upsilon_n$
```

**Example** – In our running example, the SQL query translated from SPARQL query1 is built be combining the SELECT, FROM, and WHERE clauses previously constucted. This SQL query is shown in Figure 8.6.

```
SELECT S1.RNG AS modName
FROM (
    SELECT Module.moduleId AS DOM, Module.moduleName AS RNG
    FROM Module
) AS S1, (
    SELECT Module.moduleId AS DOM, Diploma.diplomaId AS RNG
    FROM Module, Diploma
    WHERE Module.diplomaId = Diploma.diplomaId
) AS S2, (
    SELECT Diploma.diplomaId AS DOM, Diploma.diplomaName AS RNG
    FROM Diploma
) AS S3
WHERE  S1.DOM = S2.DOM
AND    S2.RNG = S3.DOM
AND    S3.RNG = "BDIA"
```

FIGURE 8.6: SQL query resulting from the translation step

### 8.3.3 Simplification of Translated SQL Query

The goal of this process is to rewrite the translated SQL query into a simplified query which is more readable to users. We think that the simplified query does better performance than the translated one since this step reduces the number of relational operations and avoid unnecessary conditions. Basically, the simplification process relies on flattening the bracketed SELECT statements used in the FROM clause of the translated SQL query.

In the following, we will use the terms "real table" and "real column" to refer to the names of tables and columns as they are in the database, e.g. module.moduleID. We will also use the term "aliased column" to refer to the names of columns given by concatenating the alias of an SQL statement (e.g. S1, S2, $\cdots$) with an alias `DOM` or `RNG`. For example, `S1.DOM` is an aliased column.

We also use the terms "inner clauses" and "outer clauses" to distinguish the clauses of SQL statements and the translated SQL query, respectively.

The whole simplification process consists of the following steps:

1. Form an auxiliary mapping of columns.
2. Move real tables from inner FROM clauses to the outer FROM clause.
3. Move conditions from inner WHERE clauses to the outer WHERE clause, assembled using AND operator.
4. Replace each aliased column in the outer SELECT clause by its corresponding real column, using the auxiliary columns mapping.
5. Replace each aliased column in the outer WHERE clause by its corresponding real column, using the auxiliary columns mapping.
6. Remove trivial/unnecessary conditions
7. Remove redundant conditions.

## Step 1

The first step of the simplification process is to establish an auxiliary mapping $CM$ between aliased columns and real columns. This mapping serve to replace aliased columns in the translated query with their corresponding real columns in the simplified query.

For each item of the outer FORM clause $FI_{out}$, we get the alias $SA$ and the SQL statement $subSQL$. In this statement, for each inner SELECT item $SI_{in}$, we get the real column name $rc$ and the alias (either `DOM` or `RNG`). The concatenation of the outer alias $SA$ and the inner alias (`DOM` or `RNG`) forms the aliased column $ac$. The aliased column $ac$ is associated with the real column $rc$ and this association is added to the auxiliary mapping.

In our running example, the translated SQL query (Figure 8.6) contains three items in the outer FROM clause. These items are aliased `S1`, `S2` and `S3` respectively. In the SQL statement of the first item, there are two columns in the inner SELECT clause:

```
SELECT Module.moduleId AS DOM, Module.moduleName AS RNG
FROM Module
```

The real column `Module.moduleId` is associated with the aliased column `S1.DOM`, and the real column `Module.moduleName` is associated with the aliased column `S1.RNG`. The following table shows the auxiliary column mapping for the example query.

| | | |
|---|---|---|
| `S1.DOM` | $\leftrightarrow$ | `Module.moduleId` |
| `S1.RNG` | $\leftrightarrow$ | `Module.moduleName` |
| `S2.DOM` | $\leftrightarrow$ | `Module.moduleId` |
| `S2.RNG` | $\leftrightarrow$ | `Diploma.diplomaId` |
| `S3.DOM` | $\leftrightarrow$ | `Diploma.diplomaId` |
| `S3.RNG` | $\leftrightarrow$ | `Diploma.diplomaName` |

### Step 2

All real tables encountered in inner FROM clauses are added to the FROM clause of the final SQL statement. Each table must appear only once.

In our running example, the distinct real tables mentioned in the inner FROM clauses are `Module` and `Diploma`. These tables form the FROM clause of the final SQL query.

### Step 3

All conditions found in inner WHERE clauses are added to the WHERE clause of the final SQL query. These conditions are assembled using AND operator.

In our running example, there is one condition in inner WHERE clauses, which is: `Module.diplomaId` = `Diploma.diplomaId`. This condition is added to the WHERE clause of the final SQL query.

### Step 4

Aliased columns found in the outer SELECT clause are replaced by their corresponding real columns from the auxiliary column mapping, and added to the final SELECT clause.

In our running example, the outer SELECT clause contains one aliased column `S1.RNG` which has `modName` as alias. This aliased column is associated with the real column `Module.moduleName`. This real column (along with the alias `modName`) forms the SELECT clause of the final SQL query.

### Step 5

Aliased columns found in the outer WHERE clause are replaced by their corresponding real columns from the auxiliary column mapping, and added to the final WHERE clause.

In our running example, the outer WHERE clause contains three conditions: (`S1.DOM = S2.DOM`), (`S2.RNG = S3.DOM`), and (`S3.RNG = "BDIA"`). By replacing the aliased columns with their corresponding real columns, we get: `Module.moduleId = Module.moduleId`, `Diploma.diplomaId = Diploma.diplomaId`, and `Diploma.diplomaName = "BDIA"`. These three conditions are added to the WHERE clause of the final SQL query (assembled using AND operator).

### Step 6

In the final WHERE clause, trivial/unnecessary conditions are removed.

In our running example, the conditions `Module.moduleId = Module.moduleId` and `Diploma.diplomaId = Diploma.diplomaId` are not necessary, thus they are removed.

### Step 7

In the final WHERE clause, any redundant conditions are removed. In our running example, there is no redundant example.

The final SQL query is shown in Figure 8.7:

```
SELECT Module.moduleName AS modName
FROM   Module, Diploma
WHERE  Module.diplomaId = Diploma.diplomaId
AND    Diploma.diplomaName = "BDIA"
```

FIGURE 8.7: SQL query resulting from the simplification process

### Summary

In this section, we have presented our first SPARQL-to-SQL translation method. This method is based on database-to-ontology mappings that are specified as associations with SQL statements. The SQL statements of the mapping associations are involved in the translated SQL query as bracketed FROM items. Therefore, a simplification step is necessary in order to flatten these bracketed statements. The simplified query is more readable to users and does better performance because the number of relational operations is reduced and unnecessary conditions are avoided.

In the next section, we present our second SPARQL-to-SQL translation method which handles database-to-ontology mappings specified using DOML language.

## 8.4 Translation using DOML Mappings

In this section, we present a method for translating SPARQL queries into SQL query in the case where mappings are provided in the form of a DOML mapping document. This kind of mappings is presented in Section 6.3. The problem that we address in this section is:

*Having a DOML mapping document $MD$ between a database $DB$ and an ontology $O$, we want to translate a SPARQL query $Q_{SPARQL}$ over the ontology $O$ into an SQL query $Q_{SQL}$ over the database $DB$.*

In the previous section, we presented a translation method using "*Associations with SQL Statements*". In that method, each concept and property in the local ontology was associated with an SQL statement. Thus, these statements were exploited directly to form the final SQL query. However, in the case of DOML mappings, such SQL statements are not provided. Instead, mappings are expressed in the form of bridges between ontology terms (concepts and properties) and database terms (tables and columns), such bridges can be associated with conditions and transformations.

Our proposed methodology for SPARQL-to-SQL translation using DOML mappings consists of the following steps:

1. **Parsing of the DOML mapping document** – this step aims at analyzing the mapping document to retrieve its different components: database tables and columns, concept bridges, property bridges, conditions and transformations.
2. **Parsing of the SPARQL Query** – this step aims at analyzing the SPARQL query to retrieve its different components: selected variables, order variables, filter clauses, and the basic graph pattern (BGP).
3. **Preprocessing** – the goal of this step is to compute an auxiliary mapping between the variables mentioned in the SPARQL query and the concepts of the ontology. This auxiliary mapping serve to determine, for each property mentioned in the query, which is the suitable property bridge when several bridges are possible.
4. **SPARQL-to-SQL Translation** – this is the main step, where the different clauses of the SQL query (SELECT, FROM, WHERE) are constructed using the information provided in the previous steps.

### Running Example

In order to illustrate our translation method, we will use the DOML mapping document which we used as a running example in Section 6.3. This document is depicted in Appendix D.

In addition, the following SPARQL query is used to demonstrate the different steps of the translation methodology. This query looks for the names of students who study in the department named `"IEM"`.

```
SELECT ?name
WHERE {
    ?s     a               ont:Student;
           ont:name        ?name;
           ont:studies-in  ?dept.
    ?dept  ont:dept-name   ?deptName.
    FILTER (?deptName = "IEM").
}
```

FIGURE 8.8: Example SPARQL Query 1

## 8.4.1 Parsing of the DOML Mapping Document

The objective of this step is to analyze the DOML mapping document and extract from it the different mapping bridges, conditions and transformations. Since the mapping document is written in RDF format, we can use ad-hoc SPARQL queries over this document to retrieve information about the different components. This process consists of the following steps:

1. Extract columns.
2. Extract concept bridges.
3. Extract datatype property bridges.
4. Extract object property bridges.
5. Extract argument lists.

6. Extract conditions.
7. Extract transformations.

### 1. Extract Columns

We saw in Section 6.3.3, that in DOML language, database tables and columns are represented as RDF resources using the primitives `doml:Table` and `doml:Column`, respectively. The name of a table or a column is specified using the predicate `doml:name`. In addition, for each column resource, the table to which this column belongs is specified using the property `doml:belongsToTable`.

In order to retrieve these information from the mapping document, we use the following SPARQL query, that retrieves all column resources with their column names and table names.

```
SELECT ?col ?colName ?tableName
WHERE {
   ?col     a                   doml:Column;
            doml:belongsToTable  ?table;
            doml:name           ?colName.
   ?table   a                   doml:Table;
            doml:name           ?tableName.
}
```

The execution of this query, over the mapping document of Appendix D, gives the following results:

```
----------------------------------------------------------
| col                         | colName      | tableName    |
==========================================================
| map:person-salary-col       | "salary"     | "person"     |
| map:person-deptid-col       | "dept_id"    | "person"     |
| map:dept-deptid-col         | "dept_id"    | "department" |
| map:person-firstname-col    | "first_name" | "person"     |
| map:person-lastname-col     | "last_name"  | "person"     |
| map:person-year-col         | "year"       | "person"     |
| map:person-email-col        | "email"      | "person"     |
| map:person-status-col       | "status"     | "person"     |
| map:dept-deptname-col       | "dept_name"  | "department" |
----------------------------------------------------------
```

These results are encapsulated as column objects and stored in a suitable structure (actually, a hash map[2]), such that a column object can be retrieved using the column resource, also called identifier (e.g. `map:person-salary-col`).

Thus, the DOML parser offers the function `getColumnByID(`*`columnRes`*`)` that takes one parameter which is a resource representing a column (an instance of `doml:Column`), and returns the 'real' column object of this resource.

### 2. Extract Concept Bridges

We saw in Section 6.3.4.1, that in DOML language, concept bridges are represented as RDF resources using the primitive `doml:ConceptBridge`. The URI of the mapped concept is specified using the predicate doml:class, whereas the mapped table (an instance of `doml:Table`)

[2] http://en.wikipedia.org/wiki/Hash_table

is specified using the predicate `doml:toTable`. Possibly, a concept bridge may be associated with a condition. This condition (an instance of `doml:Condition`) is specified using `doml:when` predicate.

The following SPARQL query retrieves every concept bridge in the document, along with the ontology concept and the database table related using this bridge, and the associated condition if exists.

```
SELECT ?cb ?concept ?table ?cond
WHERE {
   ?cb  a                doml:ConceptBridge;
        doml:class       ?concept;
        doml:toTable     ?table.
   OPTIONAL { ?cb   doml:when    ?cond.}
}
```

The execution of this query, over the mapping document of Appendix D, gives the following results:

| cb | concept | table | cond |
|---|---|---|---|
| map:person-cb | ont:Person | map:person-table | |
| map:student-cb | ont:Student | map:person-table | map:status-stud-cond |
| map:employee-cb | ont:Employee | map:person-table | map:status-emp-cond |
| map:department-cb | ont:Department | map:department-table | |

These results are encapsulated as 'concept bridge' objects that can be retrieved using the concept URI (e.g. `ont:Person`). Thus, the DOML parser offers the function `getConceptBridge(`*`conceptURI`*`)` that takes one parameter which is a concept URI, and returns the 'concept bridge' object of this concept.

**3. Extract Datatype Property Bridges**

Datatype property bridges (Section 6.3.4.2) are represented as RDF resources using the primitive `doml:DatatypePropertyBridge`. The URI of the mapped datatype property is specified using the predicate `doml:datatypeProperty`. The concept bridge to which this property bridge belongs is specified using the predicate `doml:belongsToConceptBridge`. If this property corresponds to a column, then this column (instance of `doml:Column`) is specified using the predicate `doml:toColumn`. If the property corresponds to a transformation, then this transformation (instance of `doml:Transformation`) is specified using the predicate `doml:toTransform`. If the datatype property bridge is associated with a condition, then this condition (an instance of `doml:Condition`) is specified using `doml:when` predicate.

The following SPARQL query retrieves every datatype property bridge from the mapping document, along with its datatype property, the concept bridge to which it belongs, the corresponding column or transformation, and the associated condition, if exists.

```
SELECT ?dpb ?prop ?cb ?col ?trans ?cond
WHERE {
   ?dpb  a                            doml:DatatypePropertyBridge;
         doml:datatypeProperty        ?prop;
         doml:belongsToConceptBridge   ?cb;
   OPTIONAL { ?dpb   doml:toColumn     ?col . }
   OPTIONAL { ?dpb   doml:toTransform  ?trans.}
   OPTIONAL { ?dpb   doml:when         ?cond. }
}
```

The results of this query over the mapping document of Appendix D are:

TABLE 8.3: Results of Datatype property bridges

| dpb | prop | cb | col | trans | cond |
|---|---|---|---|---|---|
| student-name-dpb | name | student-cb | | fn-ln-concat | status-stud-cond |
| employee-email-dpb | email | employee-cb | person-email-col | | status-emp-cond |
| person-name-dpb | name | person-cb | | fn-ln-concat | |
| year-dpb | year | student-cb | person-year-col | | status-stud-cond |
| deptname-dpb | dept-name | department-cb | dept-deptname-col | | |
| person-email-dpb | email | person-cb | person-email-col | | |
| employee-name-dpb | name | employee-cb | | fn-ln-concat | status-emp-cond |
| student-email-dpb | email | student-cb | person-email-col | | status-stud-cond |
| salary-dpb | salary | employee-cb | person-salary-col | | status-emp-cond |

These results are encapsulated as 'datatype property bridge' objects that can be retrieved using the datatype property URI (e.g. `ont:name`). Thus, the DOML parser offers the function `getDPBs(`*`dpURI`*`)` that takes one parameter *dpURI* which is a datatype property URI, and returns the 'datatype property bridge' object of this datatype property.

#### 4. Extract Object Property Bridges

Object property bridges (Section 6.3.4.3) are represented as RDF resources using the primitive `doml:ObjectPropertyBridge`. The URI of the mapped object property is specified using the predicate `doml:objectProperty`. The concept bridge to which this property bridge belongs (domain concept bridge) is specified using the predicate `doml:belongsToConceptBridge`, whereas, the concept bridge to which this property bridge refers (range concept bridge) is specified using the predicate `doml:refersToConceptBridge`. The predicate `doml:join-via` indicates the relational join expression (an instance of `doml:Condition`) associated to this object property bridge. If the object property bridge is associated with a condition, then this condition (an instance of `doml:Condition`) is specified using `doml:when` predicate.

The following SPARQL query retrieves every object property bridge in the document, along with the object property, the domain concept bridge, the range concept bridge, the join expression, and the associated condition if exists.

```
SELECT ?opb ?prop ?dom ?rng ?join ?cond
WHERE {
   ?opb  a                          doml:ObjectPropertyBridge;
         doml:objectProperty        ?prop;
         doml:belongsToConceptBridge  ?dom;
         doml:refersToConceptBridge   ?rng;
         doml:join-via              ?join.
   OPTIONAL { ?opb   doml:when        ?cond. }
}
```

The result of this query over the mapping document of Appendix D are:

TABLE 8.4: Results of object property bridges

| opb | prop | dom | rng | join | cond |
|---|---|---|---|---|---|
| works-for-opb | works-for | employee-cb | department-cb | person-dept-join | status-emp-cond |
| studies-in-opb | studies-in | student-cb | department-cb | person-dept-join | status-stud-cond |

These results are encapsulated as 'object property bridge' objects that can be retrieved using the object property URI (e.g. `ont:works-for`). Thus, the DOML parser offers the function `getOPBs(`*opURI*`)` that takes one parameter *opURI* which is an object property URI, and returns the 'object property bridge' object of this object property.

## 5. Extract Argument Lists

The arguments of conditions and transformations are represented using argument lists. In an argument list (doml:ArgList), the order of arguments is specified using `rdf:_1`, `rdf:_2`, ⋯ predicates. An argument can be a constant value, a database column (`doml:Column`), a transformation (`doml:Transformation`) or a condition (`doml:Condition`).

Firstly, the following SPARQL query is used to retrieve all resources that represents argument lists from the mapping document:

```
SELECT ?arglist
WHERE {
  ?arglist   a      doml:ArgList.
}
```

For the mapping document of Appendix D, the results of this query are:

```
--------------------
| arglist          |
====================
| map:arg-list-j   |
| map:arg-list13   |
| map:arg-list12   |
| map:arglist-fnln |
--------------------
```

For each one of these results/argument lists, another SPARQL query is used to retrieve the arguments with their order (`rdf:_1`, `rdf:_2`, $\cdots$) and type (`doml:Column`, `doml:Transformation`, or `doml:Condition`). For example, for the first argument list `map:arg-list-j`, the query will be:

```
SELECT ?num ?arg ?argtype
WHERE {
  map:arg-list-j        ?num   ?arg;
  OPTIONAL { ?arg     a      ?argtype }.
}
```

The results of this query are shown below:

```
-------------------------------------------------
| num      | arg                     | argtype     |
=================================================
| rdf:_2   | map:dept-deptid-col     | doml:Column |
| rdf:_1   | map:person-deptid-col   | doml:Column |
| rdf:type | doml:ArgList            |             |
-------------------------------------------------
```

**6. Extract Conditions**

In DOML language, a condition is represented using the primitive `doml:Condition`. The condition operator (AND, OR, Equals, $\cdots$) is specified using the predicate `doml:conditionType`. The argument list (an instance of `doml:ArgList`) of the condition is specified using the predicate `doml:hasArguments`.

The following SPARQL query is used to retrieve these information from the mapping document:

```
SELECT ?cond ?type ?arglist
WHERE {
   ?cond    a                    doml:Condition;
            doml:conditionType   ?type;
            doml:hasArguments    ?arglist.
}
```

The results of this query over the mapping document of Appendix D are shown below:

```
----------------------------------------------------------
| cond                  | type            | arglist        |
==========================================================
| map:person-dept-join  | doml:Equals     | map:arg-list-j |
| map:status-emp-cond   | doml:Equals-str | map:arg-list13 |
| map:status-stud-cond  | doml:Equals-str | map:arg-list12 |
----------------------------------------------------------
```

These results are encapsulated as 'condition' objects that can be retrieved using the condition resource/identifier (e.g. `map:status-emp-cond`). Thus, the DOML parser offers the function `getCondByID(`*`condRes`*`)` that takes one parameter *condRes* which is a condition identifier, and returns the 'condition' object of this identifier.

**7. Extract Transformations**

Transformations are represented using the primitive doml:Transformation. The transformation type (Concat, Add, $\cdots$) is specified using the predicate `doml:transType`. The argument list (an instance of `doml:ArgList`) of the transformation is specified using the predicate `doml:hasArguments`. The following SPARQL query is used to retrieve these information from the mapping document:

```
SELECT ?trans ?type ?arglist
WHERE {
   ?trans    a                     doml:Transformation;
             doml:transType        ?type;
             doml:hasArguments     ?arglist.
}
```

The results of this query over the mapping document of Appendix D are:

```
-----------------------------------------------------
| trans              | type        | arglist          |
=====================================================
| map:fn-ln-concat | doml:Concat | map:arglist-fnln |
-----------------------------------------------------
```

These results are encapsulated as 'transformation' objects that can be retrieved using the transformation resource/identifier (e.g. `map:fn-ln-concat`). Thus, the DOML parser offers the function `getTransByID(`*transRes*`)` that takes one parameter *transRes* which is a transformation identifier, and returns the 'transformation' object of this identifier.

In summery, after this parsing step, the DOML parser store offers direct access to the objects of the mapping document using the following functions:

- `getColumnByID(`*columnRes*`)` – takes one parameter which is a resource representing a column (an instance of doml:Column), and returns the real column of this resource.
- `getConceptBridge(`*conceptURI*`)` – takes one parameter which is a concept URI, and returns the concept bridge of this concept.
- `getDPBs(`*dpURI*`)` – takes one parameter which is a datatype property URI, and returns the set of datatype property bridges of this datatype property.
- `getOPBs(`*opURI*`)` – takes one parameter which is an object property URI, and returns the set of object property bridges of this object property.
- `getCondByID(`*condRes*`)` – takes one parameter which is a condition resource and returns the real condition.
- `getTransByID(`*transRes*`)` – takes one parameter which is a transformation resource and returns the real transformation.

These functions are vital for the next steps.

### 8.4.2 SPARQL Query Parsing

This step is the same as the first step of the translation method using "Associations with SQL Statements" (see Section 9.5.3). It aims at parsing the SPARQL query to retrieve its different components:

1. a set of selected variables $SV$,
2. a set of order variables $OV$,
3. a set of filters $FLTR$, and
4. a basic graph pattern $BGP = (N, E)$, where N is a set of nodes representing subjects and objects, and E is a set of edges representing predicates.

In our running example, the result of the parsing step is as follows:

```
SV={?name}, OV={}
FLTR={(?dept-name="IEM")}
BGP=({?s,ont:Student,?name,?dept,?deptName},
     {(?s,ont:Student), (?s,?name), (?s,?dept), (?dept,?deptName)})
```

Figure 8.9 shows this BGP.

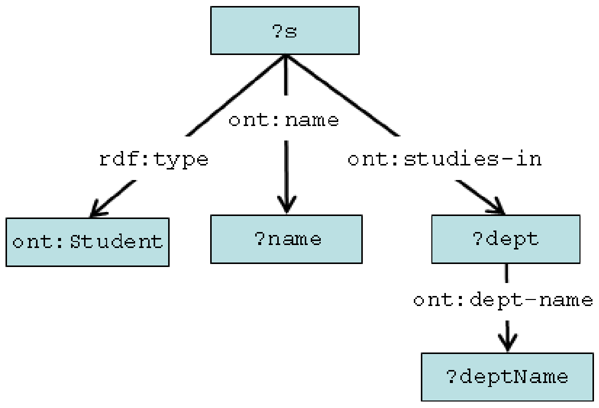


FIGURE 8.9: BGP of running example query

### 8.4.3 Preprocessing Step

In a DOML mapping document, a property (datatype or object) can have multiple property bridges. When the SPARQL query is translated to SQL, only one property bridge can be used for each property. Thus, it is necessary to determine which one of the possible bridges should be used.

For example, let us consider the following SPARQL queries:

Query 1

```
SELECT ?name
WHERE {
   ?x     ont:name       ?name.
}
```

Query 2

```
SELECT ?name
WHERE {
   ?x     rdf:type       ont:Student;
          ont:name       ?name.
}
```

Query 3

```
SELECT ?name
WHERE {
   ?x     ont:name       ?name;
          ont:works-for  ?dept.
}
```

Each of these queries mentions the datatype property `ont:name`. In the mapping document (see Appendix D), the property `ont:name` has several datatype property bridges: `person-name-dpb`, `student-name-dpb`, and `employee-name-dpb`:

```
person-name-dpb = DPB(name, person-cb, CONCAT(firstname, lastname))
student-name-dpb = DPB(name, student-cb, CONCAT(firstname, lastname),
         WHEN(person.status="Student"))
employee-name-dpb = DPB(name, employee-cb, CONCAT(firstname, lastname),
         WHEN(person.status="Employee"))
```

When we translate these SPARQL queries to SQL, we must choose one of the available datatype property bridges to use. In order to make such a decision, we compute an auxiliary mapping between the variables mentioned in the SPARQL query and the concepts of the ontology, such that when a variable $v$ is associated with a concept $C$, this means that this variable represents an instance of that concept $C$ (in the respective query).

Using this auxiliary mapping, we follow the following general strategy:

When we encounter a triple pattern of the form `{?x  P  []}`, if the property `P` has several bridges $b_1, \cdots, b_n$ that belong to the concept bridges $cb_1, \cdots, cb_n$ respectively, and if the variable `?x` is associated to a concept `C`, then we choose the property bridge that belongs to the concept bridge of the concept `C`.

In the *first query*, the only information available about the variable `?x` is that it represents the domain of the `ont:name` property, that can be `ont:Person`, `ont:Student` or `ont:Employee`. In this case, we associate the variable `?x` with the most general concept, which is `ont:Person`. Thus, in this query we use the datatype property bridge `map:person-name-dpb` because it belongs to the concept bridge of `ont:Person`.

In the *second query*, additional information about the variable `?x` is provided. That is, it is explicitly typed using the `rdf:type` predicate to be an instance of the concept `ont:Student`. In this case, we associate the variable `?x` with the concept `ont:Student`, therefore, the datatype property bridge `map:student-name-dpb` is used.

In the *third query*, no explicit typing of the variable `?x` is provided, but another triple pattern `{?x  ont:works-for  ?dept}` shares the same variable. In this triple pattern, the variable `?x` represents the domain of the object property `ont:works-for` which is `ont:Employee`. In this case, we associate the variable `?x` with the concept `ont:Employee`, and we choose to use the datatype property bridge `map:employee-name-dpb` for the datatype property `ont:name`.

The preprocessing step aims at computing the auxiliary mapping between query variables and ontology concepts. The algorithm of this step is depicted in Table H.4.

For each triple pattern whose subject node is variable, we associate this variable with a set of possible concepts. Then, for each variable, we compute the intersection of the sets of possible concepts over all triple patterns in the query. If the result of this intersection is one concept, it is associated with the variable. Otherwise, if the intersection contains more than one concept, we associate the variable with the most general one.

Let us explain this idea in more details. For each triple pattern, we distinguish three cases:

1. If the predicate node is `rdf:type`, then the variable `?x` is associated with one concept only, which is mentioned in the object node. For example, for the triple pattern `{?x rdf:type ont:Student}` we associate the variable `?x` with the concept `ont:Student`.

2. If the predicate node is a datatype property `:DP`, we get from the mapping document the datatype property bridges $dpb_1, \cdots, dpb_n$ of this property. For each of these property bridges, we get its domain concept bridge $cb_1, \cdots, cb_n$, then we get the concept of this concept bridge, the set of these concepts is associated with the variable `?x`. For example, for the triple pattern `{?x ont:name ?name}`, we get the mapping bridge of the datatype property `ont:name`, they are: `map:person-name-dpb`, `map:student-name-dpb`, `map:employee-name-dpb`. The concepts bridges to which belong these property bridges are `map:person-cb`, `map:student-cb`, and `map:employee-cb`, respectively. The concepts of these concept bridges are `ont:Person`, `ont:Student`, `ont:Employee` respectively. Thus, for this triple pattern, the variable `?x` is associated with the set `ont:Person`, `ont:Student`, `ont:Employee`.

3. If the predicate node is an object property `:OP`, we get from the mapping document the object property bridges $opb_1, \cdots, opb_n$ of this property. For each of these property bridges, we get its domain concept bridge $dcb_1, \cdots, dcb_n$, then we get the concepts of these concept bridge, the set of these concepts is associated with the variable `?x`. If the object node is also a variable `?y`, then, in addition, we get the range concept bridges $rcb_1, \cdots, rcb_n$ of the object property bridges, then we get the concepts of these concept bridge, the set of these concepts is associated with the variable `?y`. For example, for the triple pattern `{?x  ont:works-for  ?dept}`, we get the object property bridges of the object property `ont:works-for`. There is only one bridge which is `works-for-opb`. This object property bridge belongs to the concept bridge `employee-cb` (whose concept is `ont:Employee`) and refers to the concept bridge `department-cb` (whose concept is `ont:Department`). Thus, the variable `?x` is associated with the set `{ont:Employee}`, and the variable `?dept` is associated with the set `{ont:Department}`.

To this point, for each triple pattern, every variable (mentioned in this triple pattern) is associated with a set of concepts. Now, for each variable we compute the intersection of the associated sets of concepts over all the triple pattern of the query.

For example, for the query 3, in the first triple pattern `{?x ont:name ?name}`, the variable `?x` is associated with the set `{ont:Person, ont:Student, ont:Employee}`. In the second triple pattern `{?x ont:works-for  ?dept}`, the variable `?x` is associated with the set `{ont:Employee}`. The intersection of those sets is `{ont:Employee}`, thus in the query 3, the variable `?x` is associated with the concept `ont:Employee`.

Let us illustrate the preprocessing step using our running example:

In the first triple `{?s a ont:Student}`, the predicate is rdf:type, thus, the variable ?s is associated with the set {ont:Student}.

In the second triple `{?s ont:name ?name}`, the predicate is a datatype property `ont:name` that has three datatype property bridges: `map:person-name-dpb`, `map:student-name-dpb`, and `map:employee-name-dpb`. These properties belong to these concept bridges: `map:person-cb`, `map:student-cb`, and `map:employee-cb`, respectively. The concepts of these concept bridges are: `ont:Person`, `ont:Student` and `ont:Employee`, respectively. Thus for this triple pattern, the variable `?s` is associated with the set `{ont:Person, ont:Student, ont:Employee}`.

In the third triple `{?s ont:studies-in ?dept}`, the predicate `ont:studies-in` is an object property that has one object property bridge `map:studies-in-opb`. This bridge belongs to the concept bridge `map:student-cb` whose concept is `ont:Student`, and refers to the concept bridge `map:department-cb` whose concept is `ont:Department`. Thus, for this triple pattern, the variable `?s` is associated with concept `ont:Student`, and the variable `?dept` is associated with the concept `ont:Department`.

In the fourth triple `{?dept ont:dept-name ?deptName}`, the predicate is a datatype property `ont:dept-name` that has one datatype property bridge `map:deptname-dpb`. This bridge belongs to `map:department-cb` whose concept is `ont:Department`. Thus, for this triple pattern, the variable `?dept` is associated with the concept `ont:Department`.

| Triple Pattern | Associations |
|---|---|
| `?s a ont:Student` | (?s, {ont:Student}) |
| `?s ont:name ?name` | (?s,{ont:Person,ont:Student,ont:Employee}) |
| `?s ont:studies-in ?dept` | (?s, {ont:Student})<br>(?dept, {ont:Department}) |
| `?dept ont:dept-name ?deptName` | (?dept, {ont:Department}) |

Now, for the variable `?s`, the intersection of its associated sets of concepts is `{ont:Student}`. And for the variable `?dept`, the intersection is `{ont:Department}`. Therefore, in our running example, the auxiliary mapping *var2concept* contains two entries: `(?s , ont:Student)` and `(?dept, ont:Department)`.

### 8.4.4 SPARQL to SQL Translation

This is the main step of the query translation process. The input of this step is the result of the different previous steps, more precisely:

1. a parsed DOML mapping document $domlParser$,
2. a set of selected variables $SV$,
3. a set of order variables $OV$,
4. a set of filters $FLTR$,

5. a basic graph pattern $BGP = (N, E)$, and
6. an auxiliary mapping between variables and concepts *var2concept*.

The output of this step is the components of the result SQL query:

1. a set of SELECT items of the translated SQL query.
2. a set of FROM items of the translated SQL query.
3. a set of WHERE items of the translated SQL query.

The algorithm of SPARQL-to-SQL translation is depicted in Appendix H. In this algorithm, we consider some structures and functions:

- $domlParser$ – is the DOML parser that provides the following functions (see Section 8.4.1): *getConceptBridge*, *getDPBs*, *getOPBs*, *getColumnByID*, *getCondByID*, and *getTransByID*.
- *var2concept* – is the auxiliary mapping between variables and concepts, built during the preprocessing step.
- *var2column* – is an auxiliary mapping between variables and columns (also transformations), it is used to replace the variables in SELECT and FILTER clauses with suitable columns and/or SQL expressions.

The general strategy that we use to achieve the translation task consists of the following steps:

- Traverse the BGP of the SPARQL query, and treat each triple pattern of it.
- Construct the SELECT clause of the SQL query based on the variables of the SPARQL query.
- Translate the FILTER clauses of the SPARQL query into expressions in the WHERE clause of the SQL query.

### 1) Traverse the BGP

In the basic graph pattern of the query, we distinguish three kinds of triple patterns according to the predicate node:

1. Triples whose predicate is `rdf:type`, e.g. `{?x rdf:type ont:Student}`.
2. Triples whose predicate is a datatype property, e.g. `{?x ont:name ?name}`.
3. Triples whose predicate is an object property, e.g. `{?x studies-in ?dept}`.

For each edge $e$ in the BGP graph, we get its label, source vertex and destination vertex. They correspond to the predicate node, subject node, and object node of the corresponding triple pattern, respectively.

If the subject node is variable, we consider $subVar$ the name of this variable. Similarly, if the object node is variable, we consider $objVar$ the name of this variable. The predicate is an URI which is either $rdf : type$ or an ontology property.

### 1-1) The Case of `rdf:type`

The first kind of triples (triples of the form: `{?x rdf:type :C}`) is used to state that the variable `?x` is an instance of `:C`. We get the concept bridge *cb* of the concept `:C` from the mapping document (using the function `getConceptBridge` of the doml parser). From this concept bridge, we get the table and we add it to the FROM clause of the SQL query (using the procedure `addFromItem`). If the bridge *cb* is associated with a condition, then this condition is added to the WHERE clause (using the procedure `addCondition`).

### 1-2) The Case of Datatype Properties

The second kind of triples (the predicate is a datatype property) corresponds to a relationship between an instance (subject node) and the value (object node) of the specified datatype property (predicate node) for this specific instance. This value is taken from the column (or the transformation) corresponding to the datatype property.

Since a datatype property can have several datatype property bridges, we firstly looks for the suitable datatype property bridge to use. Here, we exploit the auxiliary mapping *var2concept* between variables and concepts (built in the preprocessing step). Considering that $subVar$ is the variable of the subject node, and $C$ is the concept associated to $subVar$ in the auxiliary mapping *var2concept*, thus, among the datatype property bridges of the property *dp*, we choose the bridge which belongs to the concept bridge of the concept $C$.

In other words, let $DPBs$ be the set of datatype property bridges of the datatype property *dp*, for each datatype property bridge $\beta$ in $DPBs$, let *cb* be the concept bridge to which $\beta$ belongs. If the concept $C$ of this concept bridge *cb* is the same as the concept associated with the variable $subVar$ (in the auxiliary mapping *var2concept*), then we choose $\beta$ as the suitable datatype property bridge.

Now, as we get the suitable datatype property bridge $\beta$, we check whether it refers to a column or to a transformation:

- In the first case, when $\beta$ refers to a column, we get the column *col* from the DOML parser (using the function `getColumnByID`), and we add its owner table to the FROM clause.
  - If the object node is variable, then we associate this variable $objVar$ with the column *col*. This association is put in the auxiliary mapping *var2column*.
  - Otherwise, if the object node is a literal value: *value*, then we form an expression *exp* in which the column *col* equals this literal value. This expression is added to the WHERE clause using the procedure `addWhereItem`.
- In the second case, when $\beta$ refers to a transformation, we get this transformation *trans* from the DOML parser (using the function `getTransByID`), and we translate it into an SQL expression $TS$ using the procedure `trans2SQL` (see below).
  - If the object node is variable, then we associate this variable $objVar$ with the SQL expression $TS$. This association is put in the auxiliary mapping *var2column*.

  - Otherwise, if the object node is a literal value: *value*, then we form an expression *exp* in which $TS$ equals this literal value. This expression is added to the WHERE clause using the procedure `addWhereItem`.

  For each column *col* mentioned in the transformation, we get its owner table and we add it to the FROM clause using the procedure `addFromItem`.

Finally, if the datatype property bridge $\beta$ is associated with a condition, then this condition is added to the WHERE clause of the target SQL query using the procedure `addCondition`.

The auxiliary mapping *var*2*column* is used later for replacing the variables:

- in the SELECT clause of the SPARQL query, giving the SELECT clause of the target SQL query, and
- in FILTER clauses of the SPARQL query, giving expressions that are added to the WHERE clause of the target SQL query.

#### 1-3) The Case of Object Properties

The third kind of triples (the predicate is an object property *op*) corresponds to a relationship between two instances of two concepts (domain and range of the object property) represented by two variables in the subject and object nodes of the triple.

Since an object property can have several object property bridges, we firstly looks for the suitable datatype property bridge to use. Here also, we exploit the auxiliary mapping *var*2*concept* between variables and concepts (built in the preprocessing step). Considering that $subVar$ is the variable of the subject node, and $C$ is the concept associated to $subVar$ in the auxiliary mapping *var*2*concept*, thus, among the object property bridges of the property *op*, we choose the bridge which belongs to the concept bridge of the concept $C$.

In other words, let $OPBs$ be the set of object property bridges of the object property *op* (taken from the DOML parser using the function `getOPBs`), for each object property bridge $\delta$ in $OPBs$, let *cb* be the concept bridge to which $\delta$ belongs. If the concept $C$ of this concept bridge *cb* is the same as the concept associated with the variable $subVar$ (in the auxiliary mapping *var*2*concept*), then we choose $\delta$ as the suitable object property bridge $\delta$.

Now, as we get the suitable object property bridge $\delta$, we get its associated join expression from the DOML parser using the function `getCondByID`. This join expression is added to the WHERE clause of the SQL query using the procedure `addCondition`. Tables joined by this join statement are added to the FROM clause of the target SQL query.

Finally, if the object property bridge $\delta$ is associated with a condition, then we add this condition to the WHERE clause using the procedure `addCondition`.

### 2) Construct the SELECT clause

For each variable $v$ mentioned in the SELECT clause of the SPARQL query (the set $SV$ obtained in the SPARQL query parsing step), we get its associated column/transformation from the auxiliary mapping *var*2*column* (built during the previous step). We give this column (or transformation) an alias $v$ (the variable name), and we add it to the SELECT clause. Obviously, the respective order should be kept.

## 3) Translate FILTER clauses

Similarly to the SPARQL-to-SQL translation using "*Associations with SQL Statements*" (Section 9.5.4.3) FILTER clauses are translated to equivalent SQL expressions. This translation is straightforward:

- Literal values are not changed.
- Comparing operators (=, <, <=, $\cdots$) are not changed (they exist in SPARQL and SQL).
- SPARQL logical operators (&&, ||, !) are replaced by equivalent SQL logical operators (AND, OR, NOT).
- Each variable mentioned in the FILTER is replaced by its corresponding column or transformation taken from the auxiliary mapping *var2column* (built during the previous step).

The resulting expressions are added to the WHERE clause of the target SQL query.

## 4) Miscellaneous Procedures

The procedure `addFromItem` aims at adding an item to the FROM clause of the SQL query. This procedure should guarantee that the items are distinct, thus if an item exists already in the FROM clause, it does not added again.

Similarly, the procedure `addWhereItem` aims at adding an item to the WHERE clause of the SQL query. This procedure should also guarantee that the items are distinct.

The procedure `addCondition` aims at adding a condition to the WHERE clause of the SQL query. The condition is firstly translated to an SQL expression using the function `condition2SQL` (see below). This expression is added to the WHERE clause using `addWhereItem` procedure.

In addition, for each column mentioned in the expression, the table name is added to the FROM clause using `addFromItem` procedure.

## 5) Translate Transformations

The translation of transformations is straightforward. An SQL expression is created for each transformation. The operator of this expression is taken from the transformation type, for example, `doml:Concat` becomes **"CONCAT"**, `doml:Add` becomes **"+"**, etc. The operands of the expression are taken from the arguments of the transformation, keeping the same order. According to the type of each argument, we distinguish the following cases:

- If the argument is a constant value, then the operand is this constant value.
- If the argument is a column, then the operand is the full name of this column (the concatenation of the table name and the column name).
- If the argument is another transformation, the operand is the translation of this transformation.

For example, let us consider the following transformation, that concatenates two column arguments:

```
map:fn-ln-concat    a                   doml:Transformation ;
                    doml:transType      doml:Concat ;
                    doml:hasArguments   map:arglist-fnln.
map:arglist-fnln    a                   doml:ArgList;
                    rdf:_1              map:person-firstname-col;
                    rdf:_2              map:person-lastname-col .
```

This transformation is translated to the following expression:

```
CONCAT(Person.firstName, Person.lastName)
```

where *Person.firstName* and *Person.lastName* are the full names of the columns represented by `map:person-firstname-col` and `map:person-lastname-col`, respectively.

## 6) Translate Conditions

The translation of conditions is similar to the translation of transformations. An SQL expression is created for each condition. The operator of this expression is taken from the condition type, e.g. `doml:AND` becomes **"AND"**, `doml:Equals` becomes **"="**, etc. The operands of the expression are taken from the arguments of the condition, keeping the same order. According to the type of each argument, we distinguish the following cases:

- If the argument is a constant value, then the operand is this constant value.
- If the argument is a column, then the operand is the full name of this column (the concatenation of the table name and the column name).
- If the argument is a transformation, the operand is the translation of this transformation.
- If the argument is another condition, the operand is the translation of this condition.

For example, let us consider the following condition, that restricts the column of the first argument `map:person-status-col` to the constant value of the second argument *"Student"*.

```
map:status-stud-cond    a                   doml:Condition ;
                        doml:conditionType  doml:Equals-str ;
                        doml:hasArguments   map:arg-list12.
map:arg-list12          a                   doml:ArgList ;
                        rdf:_1              map:person-status-col;
                        rdf:_2              "Student".
```

The translation of this condition is the following expression:

```
Person.status = "Student"
```

**Example** – considering our running example SPARQL query:

```
SELECT ?name
WHERE {
    ?s     a               ont:Student;
           ont:name        ?name;
           ont:studies-in  ?dept.
    ?dept  ont:dept-name   ?deptName.
    FILTER (?deptName = "IEM").
}
```

The translated SQL query is:

```
SELECT CONCAT(person.first_name, person.last_name)
  FROM department, person
 WHERE (person.status = 'Student')
   AND (person.dept_id = department.dept_id)
   AND (department.dept_name = 'IEM')
```

## 8.5 Results Reformulation

In the first section of this chapter (Section 8.1) we have presented the general procedure of query processing in data providers. The crucial step of this procedure is the SPARQL-to-SQL translation that we introduced so far in this chapter. The next step is to solve the obtained SQL query over the local database. Obviously, this step is handled by the DBMS.

Once the DBMS solves SQL query, it returns the results to the data provider. These results should be formulated in a portable and standard format before dispatching them back to the query web service.

Fortunately, SPARQL allows query results to be returned as XML, in a simple format known as the *SPARQL Query Results XML Format* [12] (see Appendix F). Therefore, we consider to use this format to express the results of the translated SQL query.

Let us consider the following SPARQL query which is submitted over the ontology shown in Figure 7.9 (this ontology is used as running example in Section 8.3). This query looks for the (first and last) names of students and in which diplomas they are enrolled.

```
SELECT ?fn ?ln ?dipName
WHERE {
   ?stud a                       ex:student;
         ex:person.firstName     ?fn;
         ex:person.lastName      ?ln.
         ex:student.diplomaID    ?dip;
   ?dip  ex:diploma.diplomaName  ?dipName;
}
```

The translation of this query into SQL yields the following SQL query [3]:

[3]This translation is done using associations with SQL statements mappings. The shown SQL query is obtained after a simplification process (Section 8.3.3).

```
SELECT person.FirstName AS fn,
       person.LastName AS ln,
       diploma.diplomaName AS dipName
  FROM student, person, diploma
 WHERE student.studentId = person.personId
   AND student.diplomaID = diploma.diplomaId
```

Let us consider that the resolution of this SQL query gives the following results:

| **fn** | **ln** | **dipName** |
|---|---|---|
| Raji | Ghawi | Master 2 3I |
| Thibault | Poulain | Master 2 3I |
| Guillermo | Gomez | Master 2 BDIA |

The formulation of these results into SPARQL query results XML format is fairly simple and regular:

1. The column names are transformed into variable names in the head section, e.g.
   `<head> <variable name="fn"/> ... </head>`
2. Each tuple is transformed into a result element (as a child of results).
3. In a tuple, each cell becomes a binding element (as a child of result), where the column name of the cell is put as the value of the name attribute of the binding, and the value of the cell is included as the content of the binding, e.g.
   `<binding name="fn"><literal>Raji</literal></binding>`

The formulation of the results of the example query are shown below:

```
<sparql xmlns="http://www.w3.org/2005/sparql-correspondances#">
  <head>
    <variable name="fn"/>
    <variable name="ln"/>
    <variable name="dipName"/>
  </head>
  <results>
    <result>
      <binding name="fn"><literal>Raji</literal></binding>
      <binding name="ln"><literal>Ghawi</literal></binding>
      <binding name="dipName"><literal>Master 2 3I</literal></binding>
    </result>
    <result>
      <binding name="fn"><literal>Thibault</literal></binding>
      <binding name="ln"><literal>Poulain</literal></binding>
      <binding name="dipName"><literal>Master 2 3I</literal></binding>
    </result>
    <result>
      <binding name="fn"><literal>Guillermo</literal></binding>
      <binding name="ln"><literal>Gomez</literal></binding>
      <binding name="dipName"><literal>Master 2 BDIA</literal></binding>
    </result>
  </results>
</sparql>
```

## 8.6 Implementation

We have implemented a prototype for each of our two SPARQL-to-SQL translation methods: 1) translation using associations with SQL statements (Figure 8.10), and 2) translation using DOML (Figure 8.11). These prototypes are written in Java language and based on several freely-available APIs, namely:

- **Jena** – we use this API for read, manipulate and write SPARQL queries.
- **Zql** – we use this API to manipulate SQL statements of mappings, and to construct the translated SQL query (before and after the simplification step).
- **JDOM** – we use this API to parse the mapping document (XML format), and to write SQL results in SPARQL query results XML format.
- **JGraphT**[4] – a free Java graph library that provides mathematical graph-theory objects and algorithms. We use this API to programmatically represent and manipulate the BGP of SPARQL queries.
- **JUNG** (Java Universal Network/Graph Framework [5]) – a free java library for the modeling, analysis, and visualization of data that can be represented as a graph or network. We use this API to visualize the BGP of SPARQL queries.
- **JDBC** – we use this interface for submitting the translated SQL query to the database and retrieve the results.

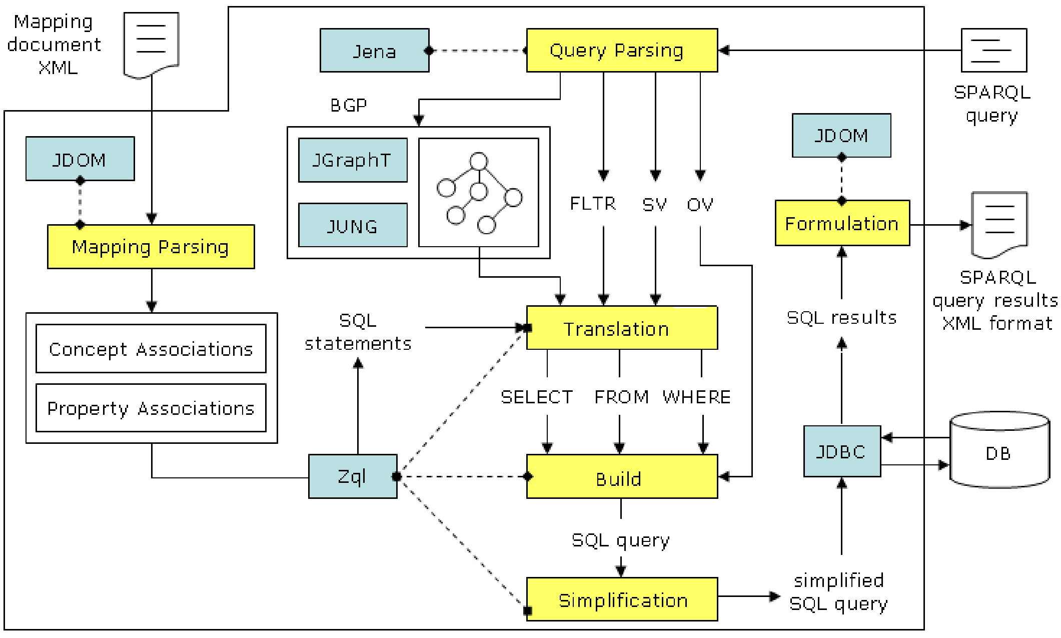


FIGURE 8.10: Implementation of the translation method for *Associations with SQL* mappings

[4]http://jgrapht.sourceforge.net/
[5]http://jung.sourceforge.net/

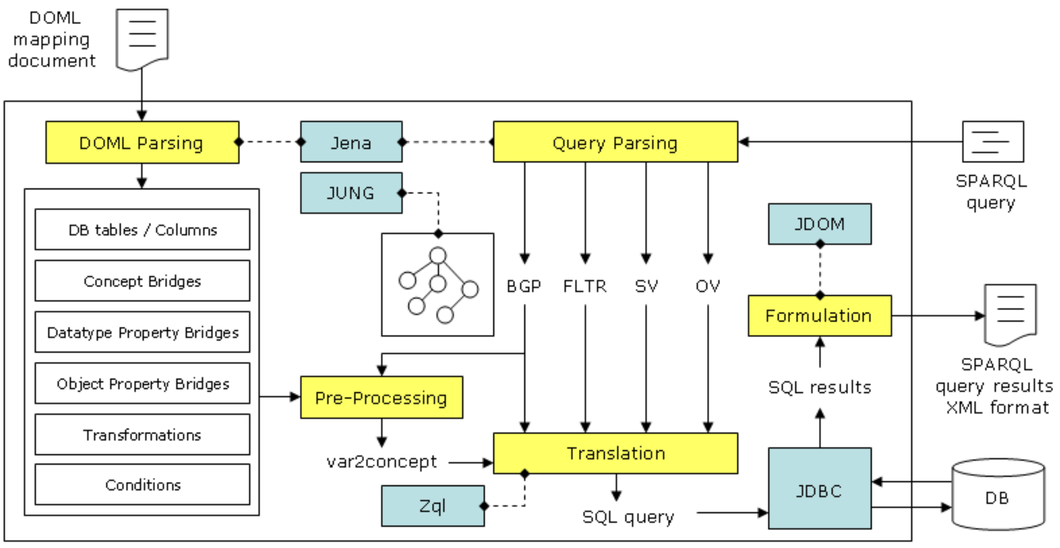


FIGURE 8.11: Implementation of the translation method for DOML mappings

## Chapter Summary

In this chapter, we presented the third task of DB2OWL tool which is the translation of SPARQL queries over an ontology into SQL queries over a mapped database.

We proposed two translation methods for our two proposed database-to-ontology mapping specifications: associations with SQL statements (Section 8.3) and DOML (Section 8.4).

In Section 8.5, we presented the process of reformulation of SQL query results into SPARQL query results XML format. In Section 8.6, we discussed about the implementation of the two proposed translation methods.

All the algorithms mentioned in this chapter can be found in Appendix H.

The next part of the thesis treats the topic of mapping XML data sources to ontologies.

Chapter 9

# Background on XML-to-Ontology Mapping

Contents

## Abstract

In Chapter 5, we gave a background on the field of database-to-ontology mapping. That background was an introduction to our contribution to this field which is presented in Chapter 7.

In this chapter, we similarly give a background on XML-to-ontology mapping which will serve as an introduction to our contribution to this field that will be presented in the next chapter.

In Appendix I, we make a brief review on XML and its related technologies, such as XML Schema. Then, we will discuss about XML as data sources and about the contexts in which XML-to-ontology mapping is needed. We give also a brief comparison between XML and ontologies.

However, in this chapter, we mainly review the literature to investigate some of existing works in this field. In particular, we are interested in approaches for 1) ontology creation from XML (Section 9.1), and 2) mapping XML to existing ontologies (Section 9.2). Finally we discuss the presented approaches.

## 9.1 Ontology Creation from XML

In this section we make a literature review to investigate some of existing approaches for ontology creation from XML. Such approaches intend to automatically transform XML Schema into newly created ontologies capturing the implicit semantics existing in the structure of XML documents.

### 9.1.1 Ferdinand et al.

Ferdinand et al. [63], propose a general binding of XML structured data to Semantic Web languages. The approach is twofold: 1) XML documents are translated into RDF graphs, and 2) XML Schemas are lifted to OWL ontologies.

**XML to RDF mapping**

Authors propose a procedure that transforms XML documents to RDF data models. In order to keep compatibility with existing documents and applications, this mapping does not require any change on either XML or RDF specifications. Two categories of XML elements are distinguished:

- elements that have sub-elements and/or attributes, and
- attributes and elements that carry only a data type value.

The mapping process starts by creating an RDF resource Document which represents the XML document itself. Then, for each subelement and attribute of the element that is currently processed (starting with the root element), an RDF property on the RDF resource is created. The value of this property is determined as follows:

1. If a datatype component (2nd category above) is encountered, then the value is represented as RDF literal on the respective property.
2. If an object component (1st category above) is encountered, then an anonymous RDF resource is created and assigned as the value of the respective property. Then, this component is processed recursively.

**XML schema to OWL mapping**

The XML schema to OWL mapping process is based on a set of interpretation and transformation rules from XML Schema to OWL. Each XML Schema complexType is mapped to an `owl:Class`. Each element and attribute declaration is mapped to an OWL property. More precisely, elements of simpleType and all attributes are mapped to an `owl:DatatypeProperty`; elements of complexType are mapped to an `owl:ObjectProperty`. Model group definitions and attribute group definitions are specializations of complex types since they only contain element respectively attribute declarations. Hence, they are also mapped to OWL classes. Two types of inheritance are supported by XML Schema: inheritance by restriction and inheritance by extension. Both are mapped to the only inheritance mechanism in OWL: `rdfs:subClassOf`.

The mappings proposed in this approach are intended to be applied for the engineering of web applications. In particular, for enhancing traditional XML languages and tools by the capabilities of OWL reasoners.

The mappings from XML to RDF, and from XML schema to OWL are independent. Therefore, the generated instances do not necessarily respect the ontology created from the XML Schema. Furthermore this approach does not tackle the question of how to create the OWL model, if no XML Schema is available.

### 9.1.2 Bohring and Auer

Bohring and Auer [27] propose a framework, called xml2owl, to create an OWL ontology from an XML schema, and convert XML data to OWL instances compliant to the created ontology. This approach is similar to Ferdinand et al. approach in the sense that it is based on a set of transformation rules from XML Schema to OWL.

In this approach, OWL classes emerge from named XSD complex Types, XSD elements containing other elements or having at least one attribute. XML nested tags are considered representing a “part-of” relationship. Thus, when an element contains another element, an OWL object property is created between their corresponding OWL classes. OWL datatype properties emerge from XML attributes and from element containing only a literal and no attributes. XML Schema arity constraints like minOccurs or maxOccurs, are mapped to the equivalent cardinality constraints in OWL, minCardinality and maxCardinality.

The mapping is implemented in XSLT and thus interoperable with different programming languages. The result of this mapping process is an OWL ontology as will as a generated XSLT stylesheet which will be used to convert XML data to OWL instances. For XML data, to which no XML schema is attached, a suitable intermediate XML schema is generated.

During the ontology creation from an XML schema, the user has no control on the process. That is, the user has no control on the newly created ontology which captures the implicit semantics existent in the XML Schema structure. Therefore the created ontologies are quite primitive, that is, they are not really semantically richer than the mapped XML Schemas.

### 9.1.3 Aninic et al.

In [5], Anicic et al. present a general method to transform an XML Schema business document specification into a rational and consistent OWL representation. The proposed method considers the specific Naming and Design Rules adopted for an XML Schema-based data exchange specification.

The mapping formalism is based on RDF-based production rules proposed in the superimposed metamodel (SIM) approach [30]. The SIM metamodel allows representation of information in a uniform way by using RDF to develop a mapping formalism that can transform information from one information representation (model) to another.The SIM metamodel provides a basic set of abstractions for uniform description of the three levels of information: model, schema, and instances. The SIM metamodel, with all three levels of information represented explicitly, enables flexible definition of transformation between any two layers; e.g., model to schema, model to model, and schema to instance [30].

Authors give two representations for the XML schema model and the OWL model in terms of the superimposed metamodel. Mapping rules are then defined between those two representations as follows:

- The XML Schema type definitions (either simple or complex) are mapped to OWL classes. Both simple and complex type names are used for the corresponding class name. SimpleType that is refinement of Datatype, has a functional datatype property that represents the literal value of that SimpleType.
- The extension and restriction of base type definitions are mapped to OWL subClassOf relationships.
- Global element declarations are mapped to OWL classes.
- Local element declarations are mapped into object properties. Element constraints (e.g., type, maxOccurs, minOccurs) are mapped to the corresponding class constraints. The corresponding class is a class defined for a complexType definition which contains the particular inner element.
- Attributes are mapped to OWL functional properties. The type of property (object or datatype) depends on the definition of an attribute (e.g., SimpleType or XML Schema predefined Datatype).
- Model groups (all, group, choice, and sequence) are mapped to hierarchies of properties.

These mapping rules can be implemented using XSLT (assuming XML representation for both models) or some rule based engine such as Jess [1]. The proposed methodology is general and can be applied to any document architecture by following the same steps and devising new transformation rules to satisfy the target document design specifications.

### 9.1.4 Xiao and Cruz

Xiao and Cruz [168][48], propose an approach to integrate heterogeneous XML sources using an ontology-based mediation architecture. The ontology integration process contains two steps: schema transformation and ontology merging (ontology mapping techniques are used).

[1]http://herzberg.ca.sandia.gov/jess/

In the first step, RDFS is used to model each XML source as a local RDF ontology to achieve a uniform representation basis for the ontology merging step. The transformation from XML to RDF is based on the following correspondences between XML Schema concepts and RDF Schema concepts:

- Complex-type elements are transformed to rdfs:Class.
- Attributes and simple-type elements are transformed to rdfs:Property.
- Element-subelement relationship is encoded as a class-to-class relationship using a new defined RDFS predicate `rdfx:contain`. Thus a new property (named `rdfx:contain`) is added whose domain being the class converted from the parent element, and its range being the class converted from the subelement.

In the second step, all the local RDF ontologies are merged using PROMPT approach [127] to generate the global ontology. During the merging process, a mapping table is produced to contain the mapping information between the global RDF ontology and local RDF ontologies.

### Summary

In this section, we have reviewed some existing approaches for ontology creation from XML data sources. The main distinction between these approaches concerns the mechanism of transforming XML constructs into ontology constructs.

The majority of approaches are based on mapping rules that directly indicate which ontology construct will be created from which XML construct. The approaches of Ferdinand et al., Bohring and Auer, Garcia and Celma, and Xiao and Cruz follow this method.

The approach of Aninic et al. is slightly different. XML schema model and OWL model are represented in terms of the superimposed metamodel, and the mapping rules are defined between the representations of the XML schema model and OWL model in terms of the superimposed metamodel.

In the next section, we review some existing approaches for mapping XML data sources to existing ontologies.

## 9.2 Mapping XML to Existing Ontology

### 9.2.1 Rodrigues et al.

Rodrigues et al. [140] propose an approach to manually map XML schemas to existing OWL ontologies with the purpose of automatically transform instances of the mapped schema into instances of the ontology. The notation used to specify mappings between XML and OWL supports several kinds of mappings: one-to-one, one-to-many, many-to-one and many-to-many. This allows mappings from one XML node to several OWL concepts and mappings from several XML nodes to one OWL concept. There are three types of mappings:

- **Class mapping** – Maps an XML node to an OWL concept
- **Datatype property mapping** – Maps an XML node to an OWL datatype property

- **Object property mapping** – Relates two class mappings to an OWL object property.

In these mappings OWL resources (classes, object and datatype properties) are addressed using their URI references, and mapped XML nodes are addressed using XPath expressions.

This approach is implemented as a mapping framework called JXML2OWL. This framework supports manual mappings from XML, XSD or DTD documents to an OWL ontology,thus supporting all the kinds of mappings such as many-to-many. According to the mapping performed, JXML2OWL generates mapping rules wrapped in an XSL document. These mapping rules allow the automatic transformation of any XML data, that is, any XML document validating against the mapped schema, into instances of the mapped ontology.

JXML2OWL has been successfully employed to integrate disparate e-tourism data sources in XML format as individuals of an e-tourism OWL ontology.

### 9.2.2 Kunfermann and Drumm

Kunfermann and Drumm [107] propose a semi-automatic approach to map XML to existing ontologies. A *Concept Finder* algorithm is proposed which is able of identifying mappings between elements of a “source” XML schema and concepts in a “target” ontology.

Creating a mapping between a source schema and a target ontology consists of three steps:

1. The user selects the source schema $S_S$ and the target ontology $O_T$. The Concept Finder algorithm parses the two files and creates an internal representation of them.
2. The user selects a seed node in $S_S$ and a matching seed concept in $O_T$ (a starting point for exploring the ontology)
3. The algorithm computes a list of mappings between $S_S$ and $O_T$.

The seed node and seed concept have to be semantically corresponding. They represent the first match of the algorithm and define the context in which the algorithm operates. The XML schema is traversed depth first and the ontology is traversed based on the matches that were found with the compared schema elements. In order to find the next match, the Concept Finder algorithm compares every child of the seed node with the relationships of the last matching concept:

- All subclasses of the seed class are compared.
- If no match is found, the algorithm compares all properties of the seed class.
- If still no match is found for the current element, the child nodes of the current element are explored in order test if a match can be derived.

This approach is semi-automatic, where the user selects a seed node in the source schema and a matching seed concept in the target ontology (a starting point for exploring the ontology). Then the Concept Finder algorithm computes a list of mappings between the source schema and the target ontology. A *similarity test* is used to determine if an element of the schema and an element of the ontology are matching based on their names.

This algorithm is evaluated using real world schemas originating from the area of B2B communication.

### 9.2.3 Kobeissy et al.

Kobeissy et al. [105] propose a mapping approach that manually creates mapping rules between XML Schema and existing OWL. These rules will be used to map XML instances to OWL individuals. The mapping does not intend to map every XML node or attribute to a class or property in the target ontology, nor to preserve the XML nesting structure. Appropriate XML nodes are chosen and mapped to ontology concepts. The mapping can be directly applied without any addition or alternation to either XML or OWL.

XML schema is used for constructing mapping rules. Information, provided by the XML Schema, about the structure and syntax of valid XML documents, are used to define mapping rules from XML elements and attributes to concepts defined by the ontology. These mapping rules are applied to XML instance files that validate the XML Schema to generate instances of the ontology.

XPath expressions are used to select XML nodes. An XPath expression returns a set of XML nodes matching. For each match, either an individual of the mapped OWL class is created for class mapping or an element/attribute content is returned for datatype properties mapping.

In this approach, mappings are represented using notations similar to those used in Rodrigues et al. approach. Class mappings relate OWL classes (identified using a Class ID) to elements in the XML Schema (represented using XPath expressions). For each XML node matching the indicated XPath expression, an individual of the mapped OWL class is created. An object property mapping relates an object property (identified using a property ID) to its domain and range classes (defined using their class IDs). Datatype property mapping defines mappings from one or more XML node to an OWL datatype property. XPath expressions are used to select the content of an element/attribute from the XML document. A datatype property relates an individual of a given class in the ontology to a value generated from the XML instances document. This approach is applied in a context management framework. It is integrated in this framework as part of an acquisition layer. The objective is to make XML context-related information retrieval seamless.

## 9.3 Discussion

In the last two sections, we have reviewed the literature to investigate existing XML-to-ontology mapping approaches. In this section we will discuss those approaches according to some common issues. The first issue to mention is the distinction between two categories of approaches. The first category of approaches aim at creating an ontology from XML/XML schema, and the second category aim at mapping XML to an existing ontology.

In both categories of approaches, we also observe two common issues that are: the used ontology language and the mode of ontology population.

Concerning the used ontology language, we note that almost all approaches use some variation of OWL language. The only exception is Xiao and Cruz approach, in which ontologies are expressed using RDFS.

Another important issue to investigate is the ontology population. Most of cited approaches follow a massive-dump ontology population where all mapped XML data are transformed into

(RDF or OWL) ontology instances. However, in some approaches such as Bohring and Auer, Kobeissy et al., and Rodrigeus et al., the generated ontology instances respect the mapped ontology schema. In Ferdinand et al. approache, the instances are not necessarily compliant with the mapped ontology. Finally, the approach of Xiao and Cruz is the only one that follow a query-driven ontology population that is based on RDQL to XQuery query translation.

Some approaches such as: Aninic et al., and Kunfermann and Drumm do not mention ontology population at all.

We also observe that there are some approaches that use XSLT mechanism for the conversion process, for example: Bohring and Auer, Aninic et al. and Rodrigues et al.

In order to achieve an efficient and complete method for building OWL ontologies from XML data sources, several aspects have to be taken in account:

1. The method should be based on XML schemas instead of documents, because an XML schema can be used by multiple documents. This will avoid generating multiple ontologies for multiple documents conforming to the same schema.
2. The method should be able to provide mapping bridges that specify the correspondences between XML entities and OWL terms. Such mapping bridges contribute into query translation between OWL and XML.
3. The method should rely on XML Schema type declarations (instead of element declarations) in order to benefit from the reusability of types by several elements within the schema. Relying on elements declarations causes generating redundant OWL terms from multiple elements of the same type.
4. XML schemas can be modeled using different design styles presented in Section I.3. Some of them use a single global element (root element), others use multiple global elements. Some styles use global types, others use only local types. Mixed styles can also be encountered. However, The mapping method should cope with all possible design patterns.
5. The method should include a finalization step that refine the generated ontology and mapping bridges. The purpose of such refinement is to adjust the structure of the generated ontology and to remove useless mapping bridges.

In the next chapter, we will present our proposed tool, called X2OWL, that aims at creating an OWL ontology from an XML data source. We will see that this tool fulfils all the points mentioned above.

## Chapter Summary

In this chapter, we have given a background on the field of XML-to-ontology mapping. We have reviewed the literature to investigate some of existing works in this field. We have presented some approaches for 1) ontology creation from XML (Section 9.1), and 2) mapping XML to existing ontologies (Section 9.2).

In Section 9.3, we have discussed the presented mapping approaches. We have concluded that we need, within OWSCIS system, a tool for ontology creation from XML, that fulfils some mentioned requirements. Our proposed tool, called X2OWL, will be presented in the next chapter.

# Chapter 10

# X2OWL Tool

**Contents**

## Abstract

In this chapter, we present X2OWL, our tool for XML-to-ontology mapping. This tool is a part of the data provider module within OWSCIS architecture. It is intended to wrap a local XML data source to a local ontology at the site level. X2OWL tool has two main tasks:

1. generate an ontology from an XML data source, and
2. translate SPARQL queries over an ontology into XQuery queries over a mapped XML data source.

Firstly, we give a general description of X2OWL (Section 10.2). Then, we present the XML-to-ontology mappings specifications used within X2OWL tool (Section 10.3). In Section 10.4, we present in details the first task of X2OWL which is: ontology generation from an XML data source. The second task of X2OWL tool, which is SPARQL-to-XQuery translation, will be presented in Chapter 11. Finally, in Section 10.6, we discuss the implementation of a prototype of X2OWL.

## 10.1 Motivation

We saw that in OWSCIS system, a participant site can contain several information sources of different types (relational databases and XML data sources). At each site, information sources are encapsulated within a data provider that aims to wrap them to a local ontology. Besides the local ontology, a data provider contains two kinds of mappings:

- mappings between the local ontology and the global ontology, and
- mappings between the local ontology and local information sources.

However, the functionality of the data provider differs according to the number and type of information sources contained in the site. Particularly, we distinguish two cases:

- The site contains a single database.
- The site contains a single XML data source.

For the first case, we have seen in the previous part of this dissertation that DB2OWL tool is deployed in the data provider to wrap a single relational database. That is, this tool handles the tasks of :1) creating the local ontology from a relational database, 2) mapping a single relational database to an existing local ontology, and 3) translate SPARQL queries over the local ontology into SQL queries over relational database.

For the second case, when a participant site contains a single XML data source, a similar tool is needed for handling the tasks of wrapping a single XML data source to a local ontology. This tool, called X2OWL aims at:

1. creating a local ontology from a single XML data source, and
2. translating SPARQL queries over the local ontology into XQuery queries over the local XML data source.

This chapter is devoted to present X2OWL tool and the methodology it follows to create the local ontology from an XML data source. In the next chapter, we will present the second task of X2OWL which is SPARQL-to-XQuery query translation.

## 10.2 X2OWL Description

As we mentioned previously, a data provider should perform three fundamental tasks:

1. Create the local ontology.
2. Map information sources to the local ontology.
3. Process local queries.

X2OWL is a tool implemented within OWSCIS framework to handle the wrapping of single XML data sources to local ontologies. This tool is deployed inside a data provider to tackle two tasks:

1. create a local ontology from a single XML data source, and
2. translate SPARQL queries over the local ontology into XQuery queries over the local XML data source (this task will be presented in the next chapter).

The first goal of X2OWL tool is to automatically create an ontology from an XML data source. The created ontology is described in OWL-DL language. It plays the role of the local ontology within the data provider.

Since we use a non-materialized approach, the generated ontology only describes the concepts and properties but not the instances. Data instances are retrieved and translated as needed in response to user queries. During ontology generation process (see Figure 10.1), X2OWL also generates a mapping document that describes the correspondences between the components of the XML data source and those of the generated local ontology. The mapping document is used for query processing purposes. In the next section, we introduce how these correspondences are specified.

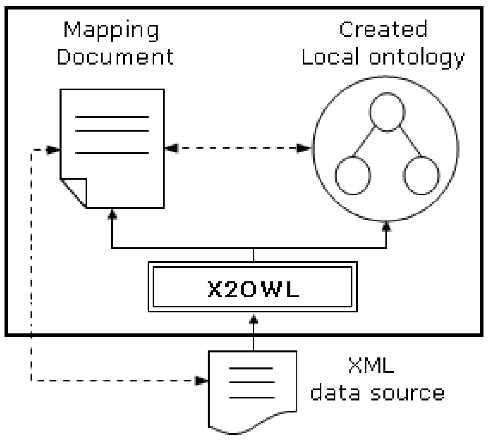


FIGURE 10.1: X2OWL Tool

In the previous chapter, we made a brief state-of-the-art on existing work about ontology creation from XML data sources. We argued that an efficient and complete method for building OWL ontologies from XML data sources should take into account several aspects, including:

1. The method should be based on XML schemas instead of documents.
2. The method should provide mapping bridges that specify the correspondences between XML entities and OWL terms.
3. The method should rely on XML schema' type declarations (instead of element declarations) in order to benefit from the reusability of types by several elements within the schema.
4. The method should cope with all possible design patterns of XML schemas (Section I.3).
5. The method should include a finalization step that refine the generated ontology and mapping bridges.

While designing our X2OWL tool, we took all these five aspects in consideration:

- Firstly, the ontology building method of X2OWL relies on the schema of the XML data source. If the XML schema does not already exist, X2OWL has the ability to automatically generate it from the source XML document.
- In addition, the building method is based on XML schema' type declarations (instead of element declarations) as we will see later in this chapter.
- During ontology generation process, X2OWL also generates a mapping document that describes the correspondences between the terms of the XML data source and the terms of the generated ontology. In section 10.3, we see how the mapping bridges can be specified.
- In order to allow our ontology generation method to cope with all possible XML schema design patterns, we define the mapping rules and algorithm in a pattern-independent fashion, as we will see in section 10.4.3.
- Finally, after the automatic ontology generation, a manual refinement step comes up for the purposes of re-structuring the generated ontology and cleaning up invalid mapping bridges, as we will see in Section 10.5.

In the next section, we introduce the XML-to-ontology mapping specification used in X2OWL tool.

## 10.3 XOML Mapping Specification

The XML-to-Ontology mapping specification that we use in X2OWL is inspired by the mapping notations used in Rodrigues et al. [140] approach (see Section 9.2.1). However, we enhanced these notations by defining a formal RDF-based syntax that is quite similar to DOML specification (see Section 6.3). Thus, we can use the primitives of DOML language that allow the definition of conditions and transformations. Consequently, we obtain a formal and rich XML-to-ontology mapping language that we call XOML (acronym for: **X**ML-to-**O**ntology **M**apping **L**anguage).

XOML language is very similar to DOML in the sense that both of them use the same structure for defining mappings in terms of concept bridges, datatype property bridges and object property bridges. Also, both allow to define conditions and transformations. However, the main difference is: the bridges in XOML relate ontology entities (addressed using their URI references) to XML nodes (addressed using their XPath expressions), whereas, in DOML, bridges relate ontology entities to database entities.

This section is dedicated to present how to specify mapping bridges in XOML language. Particularly, we introduce concept bridges, datatype property bridges and object property bridges. The other structures such as conditions and transformation are the same as in DOML language. However, we use the prefix `"xoml"` to distinguish the primitives of XOML language.

### 10.3.1 Concept Bridges

Concept Bridges are used to relate ontology concepts with XML nodes. Each concept bridge has a unique identifier within a mapping document. The primitive `xoml:ConceptBridge` is used to represent mapping bridges for ontology concepts. This primitive has three properties:

- `xoml:class` that indicates the mapped ontology concept (`rdfs:Class`), and
- `xoml:xpath` that indicates the XPath expression of the corresponding XML node.
- `xoml:when` indicates an optional condition that can be associated to the concept bridge. This condition is an instance of `xoml:Condition`. A conditional concept bridge means that this concept bridge is usable only when the associated condition holds.

Formally, in RDF representation of XOML, a concept bridge is encoded as follows:

```
[bridge-ID]  a              xoml:ConceptBridge;
              xoml:class      [class-URI];
              xoml:xpath      [xpath-exp];
              xoml:when       [condition].
```

where:

- `[bridge-ID]` is a resource representing the identifier of the concept bridge (an instance of xoml:ConceptBridge).
- `[class-URI]` is the URI of the mapped class.
- `[xpath-exp]` is a string representing the mapped XML node.
- `[condition]` is a resource representing an optional associated condition (an instance of doml:Condition), this resource should be defined somewhere in the mapping document.

For simplification, a concept bridge can be noted:

$$\boxed{id = CB(\mathbf{C},\ \mathbf{xpath},\ WHEN(condition))}$$

where *id* is the identifier of the bridge, the symbol *CB* is used to indicate that this bridge is a Concept Bridge, the first argument **C** indicates an ontology term (in this case, a concept), the second argument **xpath** indicates an XML node, and the third argument indicates an optional condition.

### 10.3.2 Datatype Property Bridges

Datatype Property Bridges are used to relate datatype properties with XML nodes. Each datatype property bridge has a unique identifier within a mapping document. A datatype property bridge belongs to exactly one concept bridge, called domain-concept-bridge (DCB), which is the concept bridge of the datatype property' domain.

A datatype property bridge can relates a datatype property to one or more XML nodes, directly or via transformations, and, possibly, with conditions.

In XOML specification, the primitive xoml:DatatypePropertyBridge is used to represent datatype property bridge. This primitive has the following features:

- `xoml:datatypeProperty` that indicates the mapped datatype property (owl:DatatypeProperty).
- `xoml:xpath` that indicates the corresponding XML node.
- `xoml:toTransform` that indicates a transformation (over one or more XML nodes) that corresponds to the mapped datatype property. This transformation is an instance of xoml:Transformation.
- `xoml:belongsToConceptBridge` that indicates the domain-concept-bridge of the datatype property bridge (should be already defined as an instance of xoml:ConceptBridge).
- `xoml:when` indicates an optional condition that can be associated to the datatype property bridge.

Formally, in RDF representation of DOML, a datatype property bridge is encoded as follows:

```
[bridge-ID]   a                            doml:DatatypePropertyBridge;
              xoml:datatypeProperty        [datatype-property-URI];
              xoml:xpath                   [xpath-exp];
              xoml:toTransform             [transformation];
              xoml:belongsToConceptBridge  [concept-bridge-id];
              xoml:when                    [condition] .
```

where:

- `[bridge-ID]` is a resource representing the identifier of the datatype property bridge (an instance of doml:DatatypePropertyBridge).
- `[datatype-property-URI]` is the URI of the mapped datatype property.
- `[xpath-exp]` is the XPath expression of the mapped XML node.
- `[transformation]` is a resource representing the transformation that corresponds to the mapped datatype property (an instance of xoml:Transformation). This resource should be defined somewhere in the mapping document.
- `[concept-bridge-id]` is a resource representing the domain-concept-bridge of the datatype property bridge (an instance of xoml:ConceptBridge), this resource should be already defined in the document.
- `[condition]` is a resource representing an optional associated condition (an instance of xoml:Condition), this resource should be defined somewhere in the mapping document.

The features `xoml:xpath` and `xoml:toTransform` are mutually exclusive.

For simplification, a direct datatype property bridge is noted:

$\boxed{id = DPB(\mathbf{dp}, \mathbf{domBrdgId}, \mathbf{xpath})}$

where:

- *id* is the identifier of the datatype property bridge,
- the symbol *DPB* is used to indicate that this bridge is a Datatype Property Bridge
- the first argument **dp** is the mapped datatype property.
- the second argument **domBrdgId** indicates the domain-concept-bridge.
- the third argument **xpath** is the XPath expression of the mapped XML node.

However, a datatype property bridge that includes a transformation is noted:

$\boxed{id = DPB(\mathbf{dp}, \mathbf{domBrdgId}, \mathbf{Trans})}$

where the third argument **Trans** indicates the corresponding transformation.

In both cases, an optional condition can be represented as a fourth argument: $WHEN(condition)$.

### 10.3.3 Object Property Bridges

An object property bridge is used to relate two concept bridges to an object property. Each object property bridge has a unique identifier within a mapping document. The two concept bridges that the object property bridge relates are respectively called:

1. *domain-concept-bridge*, which is the concept bridge of the object property' domain, and
2. *range-concept-bridge*, which is the concept bridge of the object property' range.

We say that object property bridge belongs to its domain-concept-bridge, and refers to its range-concept-bridge. Similar to other mapping bridges, object property bridges can possibly have associated conditions. In XOML language, the primitive xoml:ObjectPropertyBridge is used to represent object property bridges. This primitive has the following features:

- `xoml:objectProperty` that indicates the mapped object property (owl:ObjectProperty).
- `xoml:belongsToConceptBridge` that indicates the domain-concept-bridge of the object property bridge (should be already defined as an instance of doml:ConceptBridge).
- `xoml:refersToConceptBridge` that indicates the range-concept-bridge of the object property bridge (should be already defined as an instance of doml:ConceptBridge).
- `xoml:when` that indicates an optional condition that can be associated to the object property bridge. A conditional object property bridge means that this bridge is usable only when the associated condition holds.

In RDF representation of DOML, an object property bridge is encoded as follows:

```
[bridge-ID]  a                             xoml:ObjectPropertyBridge;
              xoml:objectProperty            [object-property-URI];
              xoml:belongsToConceptBridge    [domain-cb-id];
              xoml:refersToConceptBridge     [range-cb-id];
              xoml:when                      [condition].
```

where:

- `[bridge-ID]` is a resource representing the identifier of the object property bridge (an instance of doml:ObjectPropertyBridge).
- `[object-property-URI]` is the URI of the mapped object property.
- `[domain-cb-id]` and `[range-cb-id]` are two resources representing the domain-concept-bridge and the range-concept-bridge, respectively, of the object property bridge (both are instances of doml:ConceptBridge), these resources should be already defined in the document.
- `[condition]` is a resource representing an optional associated condition (an instance of doml:Condition), this resource should be defined somewhere in the mapping document.

For simplification, an object property bridge is noted:

$\boxed{id = OPB(\mathbf{op}, \mathbf{dcb\text{-}Id}, \mathbf{rcb\text{-}Id}, WHEN(condition))}$

where:

- *id* is the identifier of the object property bridge,
- the symbol *OPB* is used to indicate that this bridge is an Object Property Bridge,
- the first argument **op** is the mapped object property,
- the second argument **dcb-Id** indicates the domain-concept-bridge,
- the third argument **rcb-Id** indicates the range-concept-bridge,
- the fourth argument indicates an optional condition expression.

### 10.3.4 Conditions and Transformations

The description of conditions and transformations using XOML is quite similar to their description using DOML (Section 6.3.5). However, in XOML, the arguments of conditions and/or transformations can refer to XPath expressions (rather than database columns in DOML).

In the next section, we present the first task of X2OWL tool, which is ontology generation from an XML data source.

## 10.4 Ontology Generation from XML Data Source

The first task of X2OWL tool is to automatically create an ontology from an XML data source. The created ontology is described in OWL-DL language. Since we use a non-materialized approach, the generated ontology is instance-free and only contains the description of concepts and properties.

The ontology generation process relies on the XML schema of the data source. If the schema is not already provided, it can be generated from the XML document.

Having an XML schema, X2OWL can automatically generate an OWL ontology from this schema. This process is based on some mapping rules that indicates how to convert each component of the XML schema to a semantically corresponding ontology component.

During the ontology generation process, X2OWL also generates a mapping document that describes the correspondences between the database and the generated local ontology. This mapping document is expressed using XOML mapping specification presented in the previous section.

### 10.4.1 Running Example

In order to illustrate the ontology generation rules and algorithm, we use the following XML schema as running example.

```
<xsd:element name="purchaseOrder" type="PurchaseOrderType" />
<xsd:complexType name="PurchaseOrderType">
  <xsd:sequence>
    <xsd:group ref="ShipAndBill" />
    <xsd:element maxOccurs="unbounded" ref="item" />
  </xsd:sequence>
  <xsd:attribute name="orderDate" type="xsd:date" />
</xsd:complexType>
<xsd:complexType name="Address">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string" />
    <xsd:element name="street" type="xsd:string" />
    <xsd:element name="city" type="xsd:string" />
  </xsd:sequence>
  <xsd:attribute name="country" type="xsd:string" />
</xsd:complexType>
<xsd:group name="shipAndBill">
  <xsd:sequence>
    <xsd:element name="shipTo" type="USAddress" />
    <xsd:element name="billTo" type="Address" />
  </xsd:sequence>
</xsd:group>
<xsd:element name="item">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="productName" type="xsd:string" />
      <xsd:element name="price" type="xsd:decimal" />
    </xsd:sequence>
    <xsd:attributeGroup ref="ItemDelivery" />
  </xsd:complexType>
</xsd:element>
<xsd:attributeGroup name="ItemDelivery">
  <xsd:attribute name="weightKg" type="xsd:decimal" />
  <xsd:attribute name="shipBy" type="xsd:string" />
</xsd:attributeGroup>
<xsd:complexType name="USAddress" >
  <xsd:complexContent>
    <xsd:extension base="Address">
      <xsd:sequence>
        <xsd:element name="state" type="xsd:string" />
        <xsd:element name="zip" type="xsd:positiveInteger" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

### 10.4.2 Ontology Generation Rules

Our proposed method is based on XML schema, that is, the entities of schema is transformed to OWL entities. Basically, OWL Classes emerge from complex types, group declarations, and attribute declarations. Object properties emerge from element-subelement relationship. Datatype properties emerge from attributes and from simple types.

#### 10.4.2.1 Rules for Concepts

**1) Complex types**

We can distinguish two kinds of complex types:

1. global, named complex types, and
2. local anonymous complex types.

Both cases are mapped to OWL classes. However, a class generated from a global named type will have the name of that type, while a class generated from local anonymous type will have the name of its surrounding element.

For example, in our running example, there are three global complex types, named: **PurchaseOrderType**, **Address**, and **USAddress**, and one local complex type defined inside the element **item**. Therefore, for these complex types, we create four OWL classes, named: `ex:PurchaseOrderType`, `ex:Address`, `ex:USAddress`, and `ex:Item`.

**2) Element groups and attribute groups** Attribute group and element group declarations are mapped to OWL classes.

In our running example, we have one attribute group **ItemDelivery**, and one element group **shipAndBill**. Thus, for these groups, we create two OWL classes, named `ex:ItemDelivery` and `ex:ShipAndBill`.

**3) Inheritance** XML schema supports two types of inheritance: extension and restriction. Both of these inheritance mechanisms are translated to the class inheritance mechanism of OWL, using rdfs:subClassOf. When a complex type is defined as an extension or a restriction of another base complex type, then the class corresponding to this type is set as subclass of the class corresponding to the base type.

In our running example, the complex type **USAddress** is defined as an extension to the base complex type **Address**. Therefore, the class `ex:USAddress` is set as subclass of the class `ex:Address`.

Figure 10.2 shows the generated ontology after the creation of classes.

#### 10.4.2.2 Rules for Object Properties

Elements (global or local) are not mapped directly to the ontology, but element containment relationships in the schema are translated as object properties in the ontology.

That is, when an element $e_1$ contains another element $e_2$, and considering that $e_1$ is of a complex type $X_1$ and $e_2$ is of a complex type $X_2$, then an object property is created such that its domain

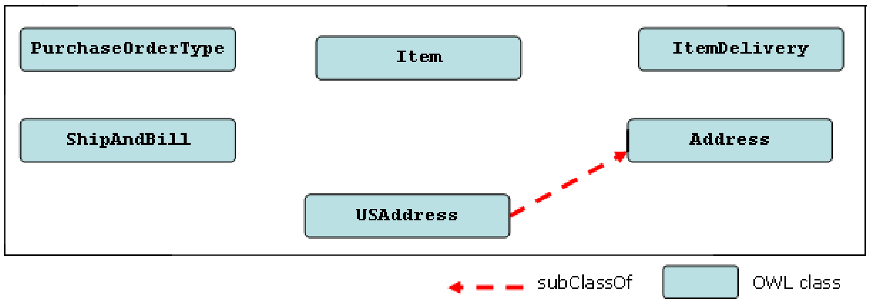


FIGURE 10.2: X2OWL - ontology generation - creation of concepts

```
<e1>
   <e2>
      ...
   </e2>
  ...
</e1>
```

```
<xs:element name="e1" type="X1" />
<xs:complexType name="X1">
  <xs:sequence>
    <xs:element name="e2" type="X2" />
    ...
  </xs:sequence>
</xs:complexType>
<xs:complexType name="X2">
  ...
</xs:complexType>
```

is the concept corresponding to $X_1$ and its range is the concept corresponding to $X_2$. The name of this object property is the concatenation of `"has"` with the name of range class.

This rule does not affected by the scope of the elements $e_1$, $e_2$ and the complex types $X_1$, $X_2$ within the schema. That is, it remains the same no matter whether anyone of the elements $e_1$, $e_2$ and complex types $X_1$ and $X_2$ are defined locally or globally.

This rule also holds for the containment of element groups and attribute groups.

In our running example, the complex type **PurchaseOrderType** contains the group **shipAndBill** and the element **item** (having a local complex type). Therefore, we create two object properties:

1. `ex:hasShipAndBill` from `ex:PurchaseOrderType` to `ex:ShipAndBill`, and
2. `ex:hasItem` from `ex:PurchaseOrderType` to `ex:Item`.

The local complex type of the element item contains the attribute group ItemDelivery. Therefore, we add an object property `ex:hasItemDelivery` from `ex:Item` to `ex:ItemDelivery`.

Finally, the element group **ShipAndBill** contains two elements **shipTo** and **billTo** that respectively have the complex types **USAddress** and **Address**. Therefore, we create two object properties:

1. `ex:hasShipTo` from `ex:ShipAndBill` to `ex:USAddress`, and
2. `ex:hasBillTo` from `ex:ShipAndBill` to `ex:Address`.

Figure 10.3 shows the generated ontology after the creation of object properties.

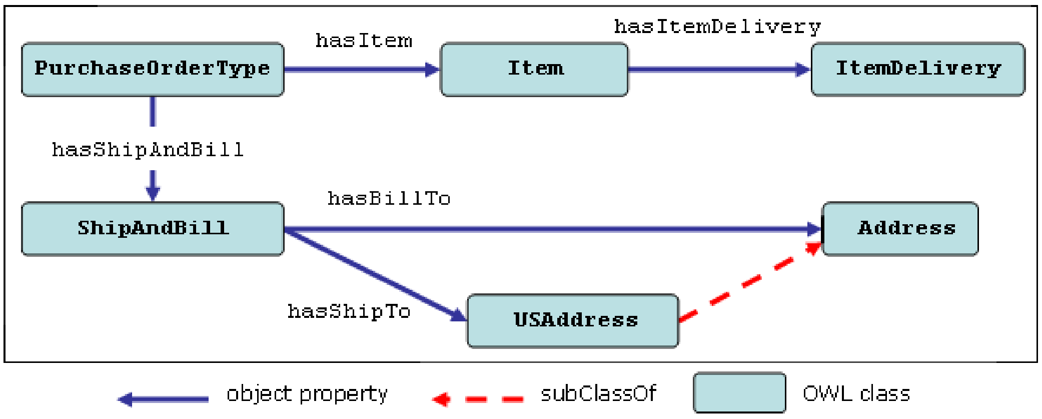


FIGURE 10.3: X2OWL - ontology generation - creation of object properties

#### 10.4.2.3 Rules for Datatype Properties

Elements of simple types are mapped to datatype properties. When a complex type *CT* (global or local) contains an element *E* of a simple type *ST* (primitive or defined), then we create a datatype property *dp* having as domain the class corresponding to the complex type *CT*. If the simple type *ST* is a primitive XSD datatype (`xsd:string`, `xsd:integer`, $\cdots$) then the range of *dp* is this datatype. Otherwise, if *ST* is defined in the schema, then the range of *dp* will be `xsd:anyType`.

In our running example, the complex type **Address** has three simple elements **name**, **street** and **city** of `xsd:string` datatype. Therefore, we create three datatype properties `ex:name`, `ex:street` and `ex:city` from `ex:Address` to `xsd:string`.

Attributes are also mapped to datatype properties. For example, the complex type **Address** has an attribute **country** of `xsd:string` datatype, therefore, a datatype property `ex:country` is created from `ex:Address` to `xsd:string`.

If a complex type is mixed, thus the elements of this type contains a text node beside the sub-elements and/or attributes. To take this text into account, a datatype property is created having as domain the class corresponding to the surrounding complex type, and having as range the datatype `"xs:string"` .

Figure 10.4 shows the genearted ontology after adding datatype properties.

### 10.4.3 Ontology Generation Algorithm

In this section, we discuss the algorithm that applies our mapping rules on the XML schema in order to generate OWL ontology entities and the suitable mapping bridges. The ontology generation process consists of the following steps:

1. XML schema parsing.
2. Generation of classes.
3. Generation of properties and mapping bridges.

The complete algorithm is depicted in Appendix J.

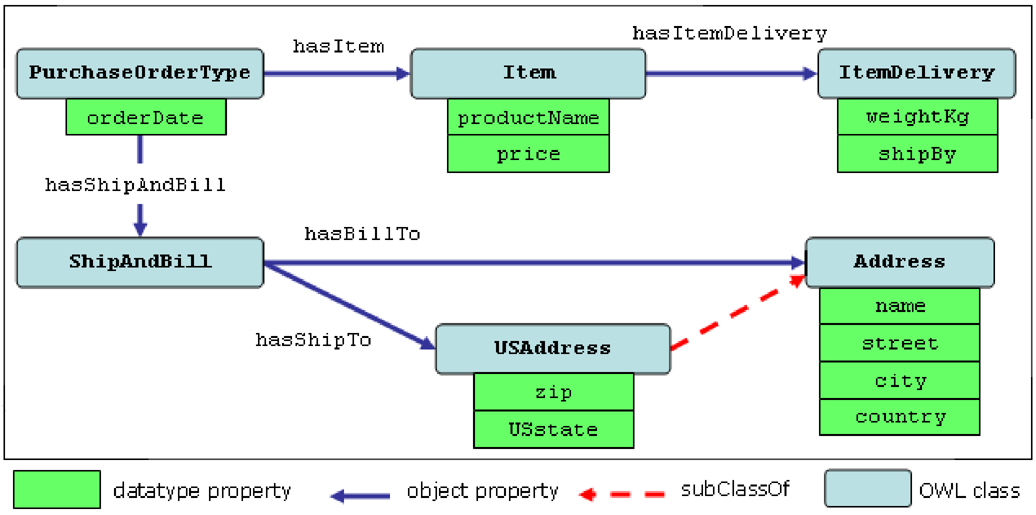


FIGURE 10.4: X2OWL - ontology generation - adding datatype properties

#### 10.4.3.1 XML Schema Parsing

The first step of ontology generation process is to parse the input XML schema. The result of this step is 1) an XML Schema Graph (XSG), and 2) an auxiliary mapping of derived complex types.

**1. XML Schema Graph (XSG)**

In order to insure that the generated ontology is independent of any specific XML schema design pattern, our algorithm is based on an XML Schema Graph (XSG) that describes the schema in the same way whatever its design style is.

An XML Schema Graph $G = (V, E)$ is generated from the XML schema, where $V$ is the vertex set, that consists of element vertices, complex type vertices, attribute vertices, element group vertices, and attribute group vertices.

$E$ is the edge set, which contains the edges established:

- from each element *el* to its complex type *ct*,
- from each complex type *ct* to its composing terms (elements, attributes, element groups, and attribute groups),
- from each element group *eg* to its composing elements.
- from each attribute group *ag* to its composing attributes.

For example, Figure 10.5 shows the XSG of the XML schema of our running example.

An XSG is a directed acyclic graph (DAG) that has always a unique root vertex which is the vertex of the root element of XML document. An XSG becomes a tree when elements and types declarations are not re-used within the schema. For example, the graph generated from a schema designed in "Russian Doll" style is always a tree.

**2. Auxiliary Mapping of Derived Complex Types**

While XML schema parsing, an auxiliary mapping **derivedTypesMap** is created that associated each derived complex type with its base (complex) type. The purpose of this mapping is to enable the setting of class hierarchy (using subClassOf) during the next step.

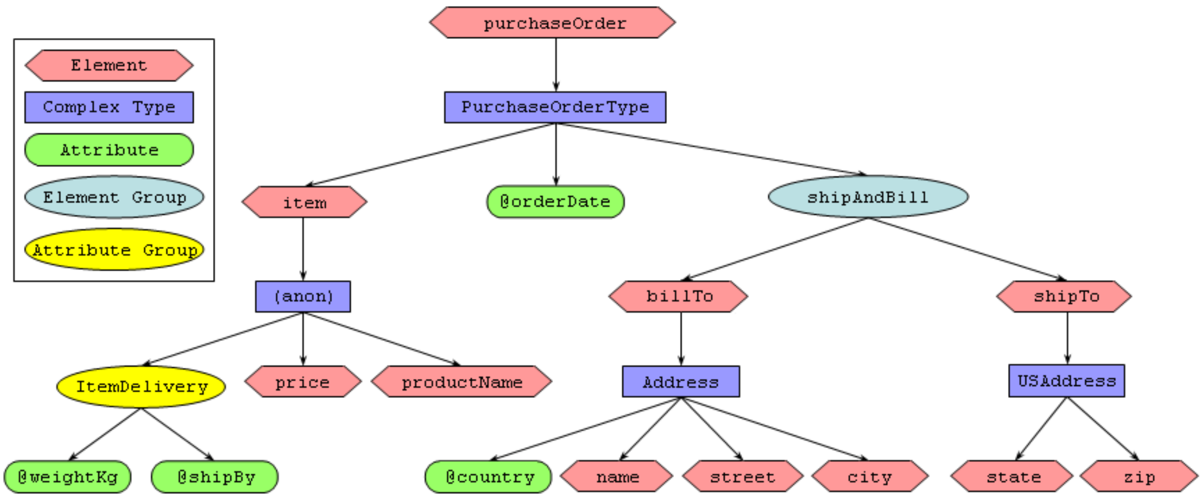


FIGURE 10.5: XSG of the running example

In our running example, this auxiliary mapping contains one entry that associates the complex type `USAddress` with its base type `Address`.

#### 10.4.3.2 Generation of Classes

In this step, OWL classes are generated according to the rules that we introduced in Section 10.4.2.1. This step is based on the **XSG** and the **derivedTypesMap** built in the previous step. That is, the XSG is traversed to retrieve complex type vertices, element group vertices, and attribute group vertices, for each one of these vertices, an OWL class is generated. Then, the derivedTypesMap is used to set up *subClassOf* relationships between classes.

Throughout, this class generation step, an auxiliary mapping **classMap** is established in order to associate generated classes with their corresponding vertices in the **XSG**.

Table J.1 depicts the algorithm of class generation.

Firstly, the vertices of the **XSG** are checked. If a vertex $v$ is a complex type vertex, element group vertex or attribute group vertex, we create a class $C$ and we add an association $(v, C)$ between the generated class $C$ and the vertex $v$ to the auxiliary mapping **classMap**.

In the case of a complex type vertex, the name of the generated class $C$ is the name of the complex type if it is global, or the name of its surrounding element (the source vertex of the incoming edge) if its local. In the case of an element group vertex (respectively, attribute group vertex), the name of the generated class $C$ is the name of that element group (respectively, attribute group).

After that, the class hierarchy is set up. To do so, for each entry in the auxiliary mapping **derivedTypesMap**, we get the vertices $v_{derivedType}$ and $v_{baseType}$ from the key and the value of this entry, respectively. These vertices correspond to the derived complex type and its base complex type, respectively. Then, from the auxiliary mapping **classMap**, we get the classes $C_{child}$ and $C_{parent}$ that are associated with the vertices $v_{derivedType}$ and $v_{baseType}$, respectively, and we set $C_{child}$ as subclass of $C_{parent}$.

In our running example, the classes generated in this step are already shown in Figure 10.2.

After the class generation step, the generation of properties and the mapping bridges comes up.

#### 10.4.3.3 Generation of Properties and Mapping Bridges

In this step, the ontology properties and the mapping bridges are generated simultaneously. This step is based on the **XSG** graph and the auxiliary mapping **classMap**.

Starting from the root vertex, the **XSG** is visited depth-first. At each vertex, a specific treatment is performed according to the kind of the vertex being visited. For the root vertex, the treatment is held by the algorithm **checkRootVertex**, whereas the rest of vertices are treated using the algorithm **checkVertex**.

The treatment of each vertex is done in a recursive manner. That is, when a vertex $v$ is visited, the algorithm **checkVertex** should be carrying some parameters about its parent vertex. After the treatment, the children of the vertex $v$ are visited and checked by the algorithm **checkVertex** that will be carrying new parameters about the current vertex $v$.

**checkRootVertex Algorithm**

As argued early, the root vertex of the **XSG** is an element vertex elroot, which corresponds to the root element of the XML data source. This root element has a complex type $CT_{root}$.

Firstly, the XPath expression of the element root $el_{root}$ is computed:

$$xpath_{root} \texttt{ = "/" + } el_{root}\texttt{.getName()}$$

Then, from the auxiliary mapping **classMap**, we get the class $C_{root}$ corresponding to the complex type $CT_{root}$ of the root element, and we create a concept bridge $CB_{root}$ between the class $C_{root}$ and the expression $xpath_{root}$.

Finally, For each one of the root children, the algorithm **checkVertex** is called with the parameters $xpath_{root}$, $C_{root}$, $CB_{root}$.

In our running example, the root vertex of the **XSG** is the vertex of the element `purchaseOrder`. The algorithm **checkRootVertex** will:

1. compute the XPath expression: $xpath_{root}$ `= /purchaseOrder`,
2. get the class $C_root = $ `ex:PurchaseOrderType` corresponding to the complex type of the element `purchaseOrder` from the auxiliary mapping **classMap**, and
3. create a concept bridge between $C_{root}$ and $xpath_{root}$:

$$CB_{root} \texttt{=(ex:PurchaseOrderType, "/purchaseOrder")}$$

**checkVertex Algorithm**

This algorithm perform specific treatment for each visited vertex according to its kind. When a vertex is checked, we carry information about its parent vertex, namely, 1) the XPath expression $xpath_{parent}$, 2) the parent class $C_{parent}$ and the 3) parent concept bridge $CB_{parent}$.

**1. Element Vertices**

When an element vertex $v_{el}$ is visited, we get the corresponding element $el$, and we compute the current XPath expression: $xpath_{el}$ `=` $xpath_{parent}$ `+ "/" +` $el$`.getName()`.

If the element $el$ has a complex type $CT_{el}$ , then :

- We get the vertex $v_{ct}$ corresponding to this complex type. It should be the (only) child of $v_{el}$, thus we get it as the destination of the only outgoing edge of $v_{el}$.
- From the auxiliary mapping **classMap**, we get the class $C_{el}$ corresponding to $v_{ct}$. we will call it the current class.
- A concept bridge $CB_{el}$ is created that associates the current class $C_{el}$ with the current XPath expression: $CB_{el}$ `= CB (`$C_{el}$`,` $xpath_{el}$`)`. We call it the current concept bridge.
- An object property $OP_{el}$ is created from the parent class $C_{parent}$ (the domain) to the current class $C_{el}$ (the range). The name of this object property is the concatenation of the prefix `"has"` with the name of the element $el$.
- An object property bridge $OPB_{el}$ is created for the object property $OP_{el}$, such that it belongs to the parent concept bridge $CB_{parent}$ (the domain concept bridge), and refers to the current concept bridge $CB_{el}$ (the range concept bridge): $OPB_{el}$ `= OPB(`$OP_{el}$`,` $CB_{parent}$`,` $CB_{el}$`)`.
- Finally, the children of the current vertex $v_{el}$ are visited and checked using **checkVertex** with the new parameters: current XPath expression $xpath_{el}$, current class $C_{el}$, and current concept bridge $CB_{el}$.

If the element $el$ has a simple type $ST_{el}$, then:

- A datatype property $DP_{el}$ is created from the parent class $C_{parent}$ (domain) to the XSD datatype of $ST_{el}$ (the range). If $ST_{el}$ is a defined simple type (not primitive), then the range is `xsd:anyType`. The name of this datatype property is the name of the current element $el$.
- A datatype property bridge $DPB_{el}$ is created for the datatype property $DP_{el}$, such that it belongs to the parent concept bridge $CB_{parent}$, and has the XPath expression $xp$ `=` $xpath_{el}$ `+ "/text()"`

  $DPB_{el}$ `= DPB(`$DP_{el}$`,` $CB_{parent}$`,` $xpath_{el}$ `+ "/text()")`

**2. Attribute Vertices**

Attributes have always simple types, therefore they are treated like elements of simple types. When an attribute vertex *vatt* is visited, we get its corresponding attribute att, and we compute the current XPath expression: $xpath_{att}$ `=` $xpath_{parent}$ `+ "/@" +` $att$`.getName()`.

Considering that $ST_{att}$ is the type of the attribute $att$:

- A datatype property $DP_{att}$ is created from the parent class $C_{parent}$ (domain) to the XSD datatype of $ST_{att}$ (the range). If $ST_{att}$ is a defined simple type (not primitive), then the range is `xsd:anyType`. The name of this datatype property is the name of the current attribute $att$.
- A datatype property bridge $DPB_{att}$ is created for the datatype property $DP_{att}$, such that it belongs to the parent concept bridge $CB_{parent}$, and has the current XPath expression $xpath_{att}$.

$DPB_{att}$ `=` `DPB(`$DP_{att}$`,` $CB_{parent}$`,` $xpath_{att}$`)`

**3. Complex Type Vertices**

When visiting complex type vertices, no treatment is performed, because they are already treated when visiting their owner elements. Therefore, when a complex type vertex $v_{ct}$ is visited, their children are soon visited, and checked using **checkVertex** with the same parameters: parent XPath expression $xpath_{parent}$, parent class $C_{parent}$, and parent concept bridge $CB_{parent}$.

**4. Element Group Vertices**

When an element group vertex $v_{eg}$ is visited, we get the corresponding element group $eg$, and we get the class $C_{eg}$ corresponding to $v_{eg}$ from the auxiliary mapping **classMap**, we call it the current class. Then:

- A concept bridge $CB_{eg}$ is created such that it associates the current class $C_{eg}$ with the parent XPath expression: $CB_{eg}$ `= CB(`$C_{eg}$`,` $xpath_{parent}$`)`. We call it the current concept bridge.
- An object property $OP_{eg}$ is created from the parent class $C_{parent}$ (the domain) to the current class $C_{eg}$ (the range). The name of this object property is the concatenation of the prefix `"has"` and the name of the element group.
- An object property bridge $OPB_{eg}$ is created for the object property $OP_{eg}$, such that it belongs to the parent concept bridge $CB_{parent}$ (the domain concept bridge), and refers to the current concept bridge $CB_{eg}$ (the range concept bridge): $OPB_{eg}$ `= OPB(`$OP_{eg}$`,` $CB_{parent}$`,` $CB_{eg}$`)`.
- Finally, the children of the current vertex $v_{eg}$ are visited and checked using **checkVertex** with the new parameters: parent XPath expression $xpath_{parent}$, current class $C_{eg}$, and current concept bridge $CB_{eg}$.

**5. Attribute Group Vertices**

When an attribute group vertex $v_{ag}$ is visited, we get the corresponding attribute group $ag$, and we get the class $C_{ag}$ corresponding to $v_{ag}$ from the auxiliary mapping **classMap**. Then, we perform the same treatment as for element groups.

Let us now illustrate this algorithm with our running example.

Considering the vertex of the element `billTo` in the XSG, when this vertex is visited, the parent parameters are:

- $xpath_{parent}$ = `"/purchaseOrder"`
- $C_{parent}$ = `ex:ShipAndBill`
- $CB_{parent}$ = `CB(ex:ShipAndBill, "/purchaseOrder")`

The visited vertex is an element vertex that has a global complex type (`Address`). Thus, the performed actions are:

- We compute the current XPath expression:

  $xpath_{el}$ `=` $xpath_{parent}$ `+ "/" + "billTo" = "/purchaseOrder/billTo"`
- get from **classMap** the (current) class corresponding to the complex type `Address`:

TABLE 10.1: Generated mapping bridges for the running example

| Concept Bridges |
|---|
| `cb1 = CB(ex:PurchaseOrderType, /purchaseOrder)` |
| `cb2 = CB(ex:item, /purchaseOrder/item)` |
| `cb3 = CB(ex:ItemDelivery, /purchaseOrder/item/@weightKg \| /purchaseOrder/item/@shipBy)` |
| `cb4 = CB(ex:shipAndBill, /purchaseOrder/billTo \| /purchaseOrder/shipTo)` |
| `cb5 = CB(ex:Address, /purchaseOrder/billTo)` |
| `cb6 = CB(ex:USAddress, /purchaseOrder/shipTo)` |
| **Datatype Property Bridges** |
| `dpb1  = DPB(ex:orderDate, cb1, /purchaseOrder/@orderDate)` |
| `dpb2  = DPB(ex:weightKg, cb3, /purchaseOrder/item/@weightKg)` |
| `dpb3  = DPB(ex:shipBy, cb3, /purchaseOrder/item/@shipBy)` |
| `dpb4  = DPB(ex:price, cb2, /purchaseOrder/item/price/text())` |
| `dpb5  = DPB(ex:productName, cb2, /purchaseOrder/item/productName/text())` |
| `dpb6  = DPB(ex:street, cb5, /purchaseOrder/billTo/street/text())` |
| `dpb7  = DPB(ex:country, cb5, /purchaseOrder/billTo/@country)` |
| `dpb8  = DPB(ex:city, cb5, /purchaseOrder/billTo/city/text())` |
| `dpb9  = DPB(ex:name, cb5, /purchaseOrder/billTo/name/text())` |
| `dpb10 = DPB(ex:street, cb6, /purchaseOrder/shipTo/street/text())` |
| `dpb11 = DPB(ex:name, cb6, /purchaseOrder/shipTo/name/text())` |
| `dpb12 = DPB(ex:state, cb6, /purchaseOrder/shipTo/state/text())` |
| `dpb13 = DPB(ex:zip, cb6, /purchaseOrder/shipTo/zip/text())` |
| `dpb14 = DPB(ex:city, cb6, /purchaseOrder/shipTo/city/text())` |
| **Object Property Bridges** |
| `opb1 = DPB(ex:PurchaseOrderType-item, cb1, cb2)` |
| `opb2 = DPB(ex:shipAndBill-billTo, cb4, cb5)` |
| `opb3 = DPB(ex:shipAndBill-shipTo, cb4, cb6)` |

$$C_{el} \texttt{ = ex:Address}$$

- create the current concept bridge:

$$CB_{el} \texttt{ = CB(}C_{el}\texttt{, } xpath_{el}\texttt{) = CB(ex:Address, "/purchaseOrder/billTo")}$$

- create an object property from the parent class to the current class:

$$OP_{el} \texttt{ = ex:hasBillTO ; ex:ShipAndBill} \longrightarrow \texttt{ex:Address}$$

- create an object property bridge for this object property, it belongs to the parent concept bridge, and refers to the current concept bridge:

$$OPB_{el} \texttt{ = OPB(ex:hasBillTO, } CB_{parent}\texttt{, } CB_{el}\texttt{)}$$

Table 10.1 shows all the generated mapping bridges for the running example.

## 10.5 Refinement

The refinement step has two purposes:

1. **Clean mapping bridges** – In some cases, the automatic nature of mapping generation causes some invalid mapping bridges. Such invalid mappings are due to the fact that different types and elements in the schema reference or share the same type and/or element. Thus, some XPath expressions that are automatically induced from the schema are not actually valid in the original XML document.

2. **Re-structure the ontology** – Humans may not admit the structure of the automatically generated ontology. The refinement step allows a human expert to modify the ontology structure manually.

In order to explain this step, we will use the following example:

### 10.5.1 Illustrative Example

Let us consider the following XML document that describes a shipment order:

TABLE 10.2: Sample XML document to illustrate refinement process

```
 1.  <shiporder orderid="889923">
 2.     <orderperson>John Smith</orderperson>
 3.     <items>
 4.        <item>
 5.           <title>Empire Burlesque</title>
 6.           <quantity>1</quantity>
 7.           <price>10.90</price>
 8.        </item>
 9.        <item>
10.           <title>Hide your heart</title>
11.           <quantity>1</quantity>
12.           <price>9.90</price>
13.        </item>
14.        <item>
15.           <title>Hearts of Fire</title>
16.           <quantity>1</quantity>
17.           <price>10.50</price>
18.       </item>
19.     </items>
20.     <ships>
21.        <ship>
22.           <date>12-01-2009</date>
23.           <item title="Empire Burlesque" />
24.           <item title="Hide your heart" />
25.        </ship>
26.     </ships>
27.  </shiporder>
```

We note that the element `<item>` is mentioned twice, but at each time the item' title is mentioned differently:

1. when the element `<item>` is mentioned as subelement of `<items>` (line 4) the title is mentioned as subelement of `<item>`:

   `<item><title>...</title>...</item>`

2. when the element `<item>` is mentioned as subelement of `<ship>` (line 23) the title is mentioned as attribute of `<item>`:

   `<item title="..." />`

However, in the XML schema of this document (Figure 10.3 ), both cases are represented by one global element declaration item (line 37), which is referenced by both `<items>` and `<ship>` element declarations. Inside this element, the title is defined twice:

1. as a element `<title>` (for representing the first case above), and

2. as an attribute `@title` (for representing the second case).

Table 10.3: XML schema for the illustrative XML document

```
<xs:element name="shiporder">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="orderperson" type="xs:string"/>
      <xs:element ref="items"/>
      <xs:element ref="ships"/>
    </xs:sequence>
    <xs:attribute name="orderid" use="required" type="xs:integer"/>
  </xs:complexType>
</xs:element>
<xs:element name="items">
  <xs:complexType>
    <xs:sequence>
      <xs:element maxOccurs="unbounded" ref="item"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="ships">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="ship"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="ship">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="date" type="xs:NMTOKEN"/>
      <xs:element maxOccurs="unbounded" ref="item"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="item">
  <xs:complexType>
    <xs:sequence minOccurs="0">
      <xs:element name="title" type="xs:string"/>
      <xs:element name="quantity" type="xs:integer"/>
      <xs:element name="price" type="xs:decimal"/>
    </xs:sequence>
    <xs:attribute name="title" type="xs:string"/>
  </xs:complexType>
</xs:element>
```

The **XSG** graph of th is XML schema is shown in Figure 10.6 (left). The ontology generated from this schema is shown in Figure 10.6 (right).

We can note that, in the generated ontology, the item' title is mentioned only once, as a datatype property `ex:title` of the class `ex:Item`. This datatype property corresponds to different entities in the schema: `<title>` element and `@title` attribute of the element `<item>`.

Figure 10.4 shows the mapping bridges established during the ontology generation process.

### 10.5.2 Mappings Cleaning

The first purpose of refinement step is to detect and remove invalid mapping bridges. Invalid mappings have to be removed because if they would be used in query resolution they would lead to invalid queries that return no results.

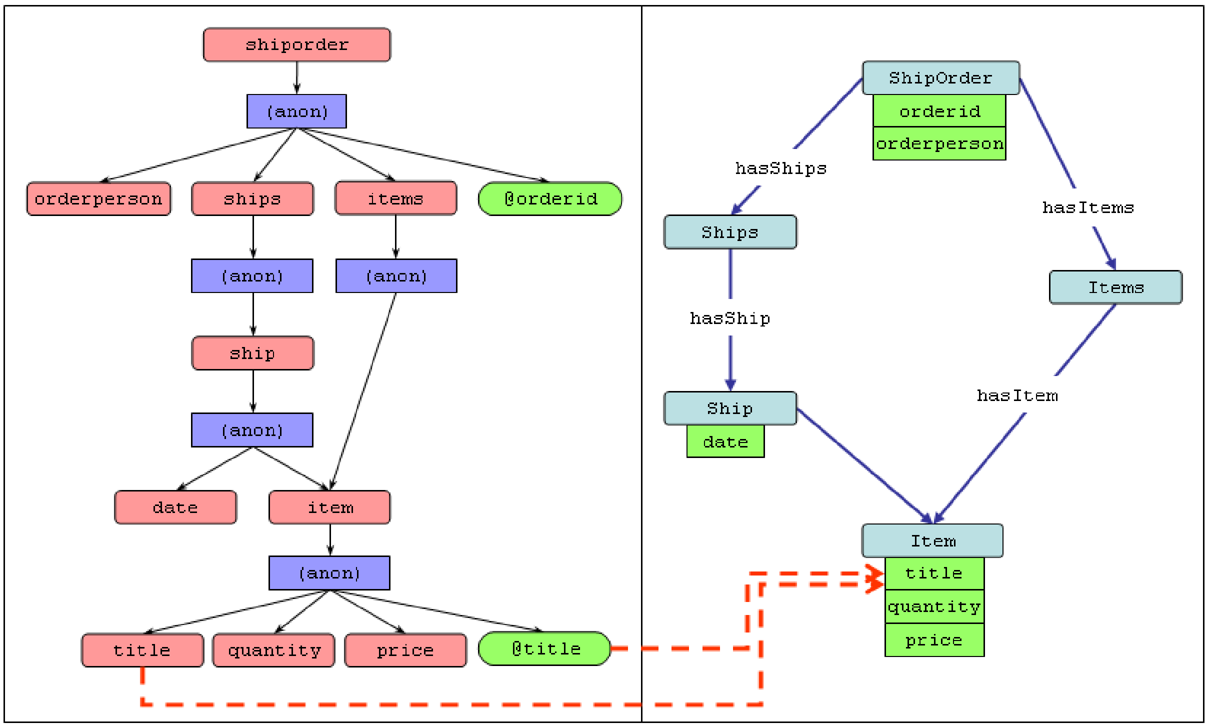


FIGURE 10.6: XSG and the generated ontology of the illustrative example

We say that a mapping bridge is invalid if:

1. it contains an invalid XPath expression (with respect to a given XML document), or
2. it references another invalid mapping bridge.

In our illustrative example, the following mapping bridges are invalid because they contain invalid XPaths with respect to the original XML document (shown in Table 10.2).

```
dpb2  = (title, cb3, /shiporder/items/item/@title)
dpb8  = (quantity, cb6, /shiporder/ships/ship/item/quantity/text())
dpb9  = (title, cb6, /shiporder/ships/ship/item/title/text())
dpb10 = (price, cb6, /shiporder/ships/ship/item/price/text())
```

Detecting invalid mappings can be done automatically if an XML document is provided which is considered representative/ typical of all XML documents conforming to the used XML schema. In this case, all possible XPath expressions of this document are extracted. Then, XPath expressions of the mapping bridges are compared with those extracted from the typical XML document. Any mapping bridge that contains an XPath expression non-belonging to the typical XML document is considered invalid. Furthermore, mapping bridges are rescanned to detect mapping bridges that reference invalid mapping bridges. Those bridges are also considered invalid. The final result of this process is a clean mapping document that only contains valid bridges.

If no typical XML document is provided, the cleaning process can be done by a human expert manually.

Table 10.4: Mapping bridges for the illustrative example

| Concept Bridges |
|---|
| `cb1 = (shiporder, /shiporder)` |
| `cb2 = (items, /shiporder/items)` |
| `cb3 = (item, /shiporder/items/item)` |
| `cb4 = (ships, /shiporder/ships)` |
| `cb5 = (ship, /shiporder/ships/ship)` |
| `cb6 = (item, /shiporder/ships/ship/item)` |

| Datatype Property Bridges |
|---|
| `dpb1 = (orderid, cb1, /shiporder/@orderid)` |
| `dpb2 = (title, cb3, /shiporder/items/item/@title)` |
| `dpb3 = (quantity, cb3, /shiporder/items/item/quantity/text())` |
| `dpb4 = (title, cb3, /shiporder/items/item/title/text())` |
| `dpb5 = (price, cb3, /shiporder/items/item/price/text())` |
| `dpb6 = (orderperson, cb1, /shiporder/orderperson/text())` |
| `dpb7 = (title, cb6, /shiporder/ships/ship/item/@title)` |
| `dpb8 = (quantity, cb6, /shiporder/ships/ship/item/quantity/text())` |
| `dpb9 = (title, cb6, /shiporder/ships/ship/item/title/text())` |
| `dpb10 = (price, cb6, /shiporder/ships/ship/item/price/text())` |
| `dpb11 = (date, cb5, /shiporder/ships/ship/date/text())` |

| Object Property Bridges |
|---|
| `opb1 = (hasItems, cb1, cb2)` |
| `opb2 = (hasItem, cb2, cb3)` |
| `opb3 = (hasShips, cb1, cb4)` |
| `opb4 = (hasShip, cb4, cb5)` |
| `opb5 = (hasItem, cb5, cb6)` |

### 10.5.3 Ontology Re-Structuring

The refinement step also includes an optional process of restructuring the generated ontology. Humans may not admit the structure of the automatically generated ontology. Our approach allows a human expert to modify the ontology structure manually. For example, he can rename or remove ontology terms, or change the domain and the range of a property. However, modifying the ontology structure necessitates appropriate modifications of mapping bridges in order to keep them consistent with the ontology.

In our illustrative example, the human expert may decide to remove the class `ex:ships` and then to relate the class `ex:shiporder` directly to the class `ex:ship` (via the object property `ex:hasShip`).

This ontology change requires three changes on the mapping bridges (see Figure 10.7):

1. Firstly, the concept bridge of the removed class should be removed, thus, the concept bridge `cb4 = CB(ships, /shiporder/ships)` is removed.
2. Then, the object property bridge of the removed object property should be removed, thus the bridge: `opb3 = OPB(hasShips, cb1, cb4)` is removed.
3. Finally, the object property bridge of the modified object property should be appropriately modified, thus, the object property bridge `opb4 = (hasShip, cb4, cb5)` becomes `opb4 = (hasShip, cb1, cb5)`.

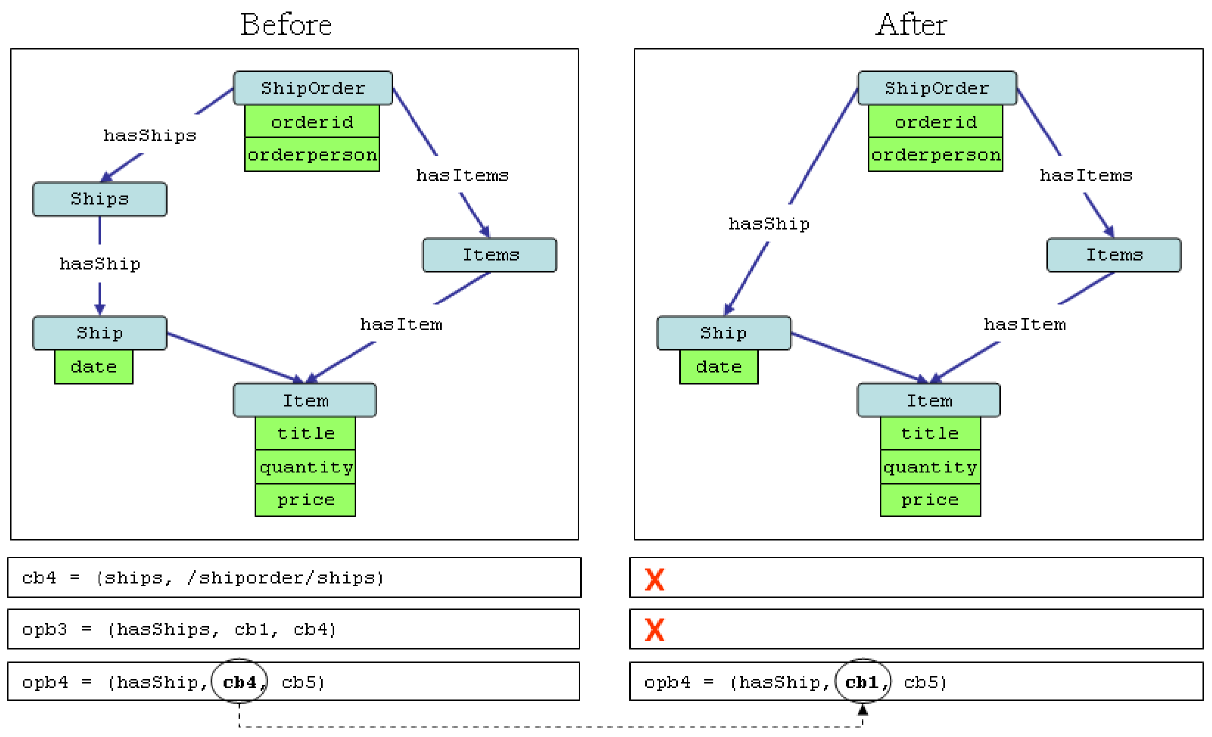

Before
After
ShipOrder
orderid
orderperson
hasShips
hasItems
Ships
hasShip
Items
Ship
date
hasItem
Item
title
quantity
price
cb4 = (ships, /shiporder/ships)
opb3 = (hasShips, cb1, cb4)
opb4 = (hasShip, cb4, cb5)
X
X
opb4 = (hasShip, cb1, cb5)

FIGURE 10.7: Example of ontology restructring and consequent mapping changes

## 10.6 Implementation

We have implemented a prototype of X2OWL tool using Java programming language. This implementation is based on several freely-available APIs for java, namely:

- **Trang**[1] – for generating XML schemas from XML documents
- **XSOM**[2] – for analyzing XML schemas,
- **JUNG**[3] – for graph-based manipulations.
- **Jena** [90] – a Java framework for building Semantic Web applications. It provides a programmatic environment for RDF, RDFS and OWL, SPARQL. We use this API for read, manipulate and write OWL ontologies.

Figure 10.8 shows hows these APIs are exploited within the implementation of X2OWL. Currently, the process of ontology restructring is not yet implemented.

## Chapter Summary

In this chapter, we have presented X2OWL, a tool for XML-to-ontology mapping. This tool has two main tasks:

1. generate an ontology from an XML data source, and

[1] http://www.thaiopensource.com/relaxng/trang.html
[2] https://xsom.dev.java.net/
[3] http://jung.sourceforge.net/

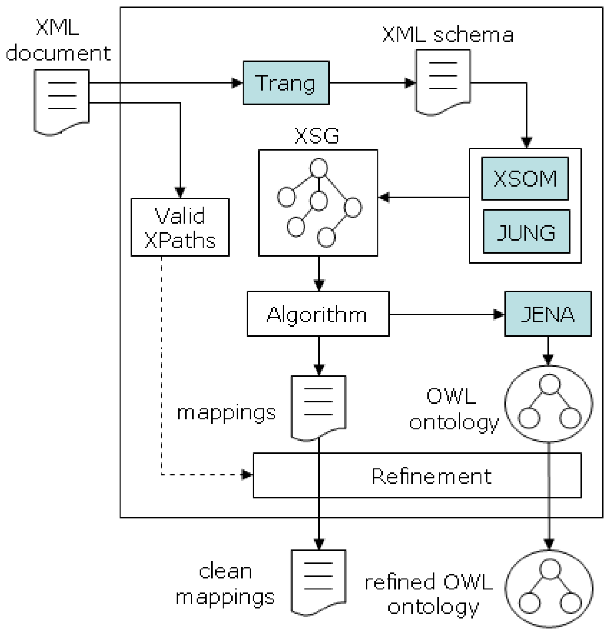


FIGURE 10.8: X2OWL implementation

2. translate SPARQL queries over an ontology into XQuery queries over a mapped XML data source.

In Section 10.3, we presented XOML, the XML-to-ontology mapping specification used by X2OWL tool.

In Section 10.4, we disscused the first task of X2OWL which is: ontology generation from a XML data source. We firstly introduced the mapping rules (Section 10.4.2). Then, we presented the ontology generation algorithm (Section 10.4.3), and the refinement process (Sectin 10.5).

We concluded this chapter by introducing an implementation prototype of X2OWL (Section 10.6).

In the next Chapter, we present the second task of X2OWL, which is SPARQL-to-XQuery tranlation.

# Chapter 11

# SPARQL-to-XQuery Translation

**Contents**

## Abstract

In this chapter, we present a method for translating a SPARQL query over an ontology into an XQuery query over a mapped XML data source. This translation process is based on a mapping document (expressed using XOML language, Section 10.3) between the ontology and the XML data source. It represents the second task of X2OWL tool presented in the previous chapter.

Our proposed SPARQL-to-XQuery translation method consists of the following steps:

1. Parsing the XOML mapping document (Section 11.2.2).
2. Parsing the SPARQL Query (Section 11.2.3).
3. Preprocessing (Section 11.2.4).
4. Building the XQuery query (Section 11.2.5).

In Section 11.3, we discuss the implementation of the translation method.

## 11.1 Motivation

We saw previously that query processing within OWSCIS system involves a phase of query translation. Queries submitted over the local ontologies (expressed in SPARQL) are translated into semantically equivalent queries over the local information sources (expressed using the native query languages of these sources, such as SQL for relational databases, and XQuery for XML data sources).

In Chapter 8, we presented the query translation process for the case of relational databases (SPARQL-to-SQL translation). In this chapter, we present the query translation process for the case of XML data source, that is, SPARQL to XQuery translation.

The SPARQL-to-XQuery translation is based on the mappings bridges between the local ontology and the underlying XML data source. These mapping bridges (expressed in XOML language) are automatically generated during the ontology creation using X2OWL tool that we introduced in the previous chapter.

In the next section, we present our query translation method in details.

## 11.2 SPARQL to XQuery Translation Algorithm

In this section, we will present our proposed method for translating SPARQL queries into XQuery queries. This method is based on XML-to-Ontology mappings that are provided in the form of an XOML mapping document (this mapping specification is presented in Section 10.3).

Mappings are expressed in the form of bridges between ontology terms (concepts and properties) and XML nodes expressed as XPath expressions. Our proposed methodology for SPARQL-to-XQuery translation using XOML mappings consists of the following steps:

1. **Parsing the XOML mapping document** – this step aims at analyzing the mapping document to retrieve its different components: concept bridges, datatype property bridges and object property bridges.

2. **Parsing the SPARQL Query** – this step aims at analyzing the SPARQL query to retrieve its different components: selected variables, order variables, filter clauses, and the triple patterns.
3. **Preprocessing** – the purpose of this step is to compute all possible mapping graphs corresponding to the basic graph of the SPARQL query. Later, only consistent mapping graphs is used to construct the target XQuery query.
4. **Building the XQuery query** – this is the main step, where the different clauses of the XQuery query (`for`, `let`, `order by`, `where`, and `return`) are constructed using the information provided in the previous steps.

### 11.2.1 Running Example

In order to illustrate our translation method, we will use the following example. We consider the XML document that we used as an illustrative example in Section 10.5.1. This document is shown in Table 10.2.

The ontology generated from this document is shown in Figure 11.1, and the XOML mapping document is depicted in Appendix K.

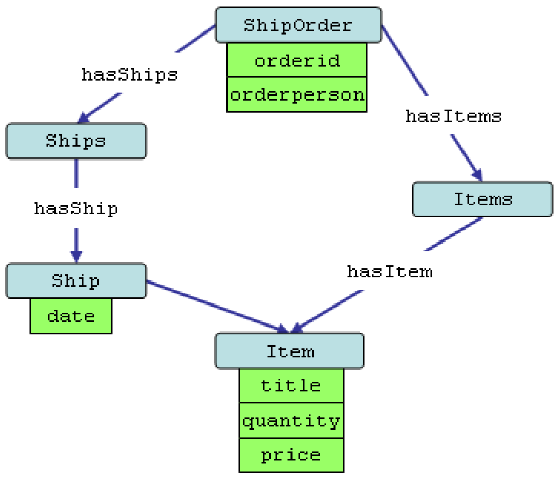


FIGURE 11.1: Ontology generated from the XML document of the running example

The following SPARQL query looks for the title of items shipped and date of shipment.

```
SELECT ?date ?title
WHERE {
   ?ship      ex:date             ?date;
              ex:ship-item        ?item.
   ?item      ex:title            ?title.
}
```

We will use this query as a running example to explain the different steps of the query translation method.

### 11.2.2 Parsing of the XOML Mapping Document

The objective of this step is to extract the different mapping bridges from the XOML mapping document. This parsing process is very similar to the process of parsing a DOML document, explained in Section 8.4.1. Since the mapping document is written in RDF format, we can use ad-hoc SPARQL queries over this document to retrieve information about the different components. This process consists of the following steps:

1. Extract concept bridges.
2. Extract datatype property bridges.
3. Extract object property bridges.

#### 1. Extract Concept Bridges

As we saw in the previous chapter, in XOML language, concept bridges are represented as RDF resources using the primitive `xoml:ConceptBridge`. The URI of the mapped concept is specified using the predicate `xoml:class`, whereas the mapped XML node (expressed as XPath expression) is specified using the predicate `xoml:xpath`.

The following SPARQL query retrieves every concept bridge in the document, along with the ontology concept and the XPath expression of this bridge.

```
SELECT ?cb ?concept ?xpath
WHERE {
  ?cb  a            xoml:ConceptBridge;
       xoml:class   ?concept;
       xoml:xpath   ?xpath.
}
```

The results of this SPARQL query over the XOML mapping document of Appendix K are:

```
---------------------------------------------------------
| cb      | concept      | xpath                        |
=========================================================
| map:cb1 | ex:shiporder | "/shiporder"                 |
| map:cb2 | ex:ships     | "/shiporder/ships"           |
| map:cb3 | ex:ship      | "/shiporder/ships/ship"      |
| map:cb4 | ex:item      | "/shiporder/ships/ship/item" |
| map:cb5 | ex:items     | "/shiporder/items"           |
| map:cb6 | ex:item      | "/shiporder/items/item"      |
---------------------------------------------------------
```

These results are encapsulated as ‘concept bridge’ objects that can be retrieved using the concept URI (e.g. `ex:item`).

Thus, the XOML parser offers the function `getConceptBridges(`*`conceptURI`*`)` that takes one parameter which is a concept URI, and returns the set of ‘concept bridges’ objects of this concept.

#### 2. Extract datatype property bridges

We saw that datatype property bridges are represented as RDF resources using the primitive `xoml:DatatypePropertyBridge`. The URI of the mapped datatype property is specified using the predicate `xoml:datatypeProperty`, whereas the mapped XML node (expressed as XPath expression) is specified using the predicate `xoml:xpath`. The concept bridge to which this property bridge belongs is specified using the predicate `xoml:belongsToConceptBridge`.

The following SPARQL query retrieves every datatype property bridge in the document, along with: 1) the datatype property URI, 2) the corresponding XPath expression, and 2) the concept bridge to which the property bridge belongs:

```
SELECT ?dpb ?prop ?cb ?xpath
WHERE {
  ?dpb  a                            xoml:DatatypePropertyBridge;
        xoml:datatypeProperty        ?prop;
        xoml:belongsToConceptBridge  ?cb;
        xoml:xpath                   ?xpath.
}
```

The results of this SPARQL query over the XOML mapping document of Appendix K are:

| dpb | prop | cb | xpath |
|---|---|---|---|
| map:dpb1 | ex:orderid | map:cb1 | /shiporder/@orderid |
| map:dpb2 | ex:orderperson | map:cb1 | /shiporder/orderperson/text() |
| map:dpb3 | ex:date | map:cb3 | /shiporder/ships/ship/date/text() |
| map:dpb4 | ex:title | map:cb4 | /shiporder/ships/ship/item/@title |
| map:dpb9 | ex:quantity | map:cb6 | /shiporder/items/item/quantity/text() |
| map:dpb10 | ex:title | map:cb6 | /shiporder/items/item/title/text() |
| map:dpb11 | ex:price | map:cb6 | /shiporder/items/item/price/text() |

**3. Extract object property bridges**

We saw that object property bridges are represented as RDF resources using the primitive `xoml:ObjectPropertyBridge`. The URI of the mapped object property is specified using the predicate `xoml:objectProperty`. The concept bridge to which this property bridge belongs (domain concept bridge) is specified using the predicate `xoml:belongsToConceptBridge`, whereas, the concept bridge to which this property bridge refers (range concept bridge) is specified using the predicate `xoml:refersToConceptBridge`.

The following SPARQL query retrieves every object property bridge in the document, along with the object property URI, the domain concept bridge, and the range concept bridge.

```
SELECT ?opb ?prop ?dom ?rng
WHERE {
  ?opb  a                            xoml:ObjectPropertyBridge;
        xoml:objectProperty          ?prop;
        xoml:belongsToConceptBridge  ?dom;
        xoml:refersToConceptBridge   ?rng;
}
```

The results of this SPARQL query over the XOML mapping document of Appendix K are:

```
-----------------------------------------------------
| opb      | prop                | dom     | rng     |
=====================================================
| map:opb1 | ex:shiporder-ships  | map:cb1 | map:cb2 |
| map:opb2 | ex:ships-ship       | map:cb2 | map:cb3 |
| map:opb3 | ex:ship-item        | map:cb3 | map:cb4 |
| map:opb4 | ex:shiporder-items  | map:cb1 | map:cb5 |
| map:opb5 | ex:items-item       | map:cb5 | map:cb6 |
-----------------------------------------------------
```

After the XOML parsing step, the XOML parser store offers direct access to the objects of the mapping document using the following functions:

- `getConceptBridges(`*`conceptURI`*`)` – takes one parameter which is a concept URI, and returns the set of concept bridges of this concept.
- `getDPBs(`*`dpURI`*`)` – takes one parameter which is a datatype property URI, and returns the set of datatype property bridges of this datatype property.
- `getOPBs(`*`opURI`*`)` – takes one parameter which is an object property URI, and returns the set of object property bridges of this object property.

The next step is to parse the SPARQL query.

### 11.2.3 SPARQL Query Parsing

Similarly to SPARQL-to-SQL translation methods (previously presented in Chapter 8), the SPARQL-to-XQuery translation should also include a SPARQL query parsing step. In this step, the SPARQL query is parsed and analyzed to get its different components, which are:

1. a set of selected variables $SV$,
2. a set of order variables $OV$,
3. a set of filters $FLTR$, and
4. a set of triple patterns $T$.

For our running example, the results of the query parsing step are:

```
SV = {?date, ?title}, OV = {}, FLTR = {}
T = { {?ship ex:date ?date},
      {?ship ex:ship-item ?item},
      {?item ex:title ?title} }
```

In the next step, the set of triples are processed in order to generate mapping graphs.

### 11.2.4 Preprocessing Step

The aim of this step is to compute a mapping graph of the SPARQL query. A mapping graph simulates the BGP graph of the SPARQL query but associates each node with an XPath expression. It is obtained by replacing each edge of the BGP by a property bridge of the ontology property of this edge.

However, since a property can have several property bridges, there will be several mapping graphs. In order to obtain an efficient SPARQL-to-XQuery translation, we should compute all these possible mapping graphs, and choose the suitable one to translate to XQuery.

A mapping graph is a directed graph $G = (N, E)$ where $N$ is a set of nodes, and $E$ is a set of directed edges. A mapping node is either:

- a variable-to-XPath association, e.g.,

`(?ship, "/shiporder/ships/ship")`

- or a literal-to-XPath association, e.g.,

  `("Hide your heart", "/shiporder/items/item/title/text()")`

The process of computing possible mapping graphs consists of two phases:

1. Firstly, for each triple pattern $t \in T$, we find the set $P^t$ of mapping pairs (edges). A mapping pair consists of two mapping nodes.
2. Then, we construct possible mapping graphs, by replacing each triple by one of its corresponding mapping pairs.
3. Finally, among possible mapping graphs, we choose the suitable one to translate to XQuery.

#### 11.2.4.1 Finding Mapping Pairs of a Triple Pattern

For each triple pattern we distinguish two cases according to the predicate node.

**Case 1**

If the predicate node is a datatype property *dp*, we get its datatype property bridges $DPBs$ from the XOML parser. For each datatype property bridge $dpb \in DPBs$, we will generate a mapping pair as follows:

- We get the concept bridge *cb* to which *dpb* belongs, then we get the XPath expression of *cb*: $xpath_{cb}$. This XPath expression is associated with the variable of the subject node $v_{sub}$. This association $vx_{from} = (v_{sub}, xpath_{cb})$ will be the start point of the target mapping pair.
- Then, we get the XPath expression $xpath_{dpb}$ of the datatype property bridge *dpb*. If the object node is a variable $v_{obj}$, then we associate $xpath_{dpb}$ with this variable $v_{obj}$. This association $vx_{to} = (v_{obj}, xpath_{dpb})$ will be the end point of the target mapping pair. Otherwise, if the object node is a literal $l_{obj}$, then we associate $xpath_{dpb}$ with this literal. This association $vx_{to} = (l_{obj}, xpath_{dpb})$ will be the end point of the target mapping pair.

Thus, the target mapping pair goes from the mapping node $vx_{from} = (v_{sub}, xpath_{cb})$ to the mapping node $vx_{to}$.

For example, in the following triple pattern `{?item ex:title ?title}`, the predicate node is a datatype property `ex:title`. This property has two datatype property bridges:

```
dpb4  = DPB( ex:title, cb4, "/shiporder/ships/ship/item/@title")
dpb10 = DPB( ex:title, cb6, "/shiporder/items/item/title/text()")
```

Thus we generate two mapping pairs:

1. The first datatype property bridge dpb4 has the XPath expression

   $xpath_{dpb_4} =$ `"/shiporder/ships/ship/item/@title"`

   and belongs to the concept bridge cb4 whose XPath is

   $xpath_{cb_4} =$ `"/shiporder/ships/ship/item"`

So, we associate the variable of the subject node ?item with the XPath $xpath_{cb_4}$ (of the concept bridge cb4) to make the start point

$vx_{from}$ `= (?item, "/shiporder/ships/ship/item")`

Then, we associate the variable of the object node `?title` with the XPath $xpath_{dpb_4}$ (of the datatype property bridge dpb4) to make the end point

$vx_{to}$ `= (?title, "/shiporder/ships/ship/item/@title")`

The first mapping pair goes from the start point $vx_{from}$ to the end point $vx_{to}$:

```
pair [from=(?item , "/shiporder/ships/ship/item"),
        to=(?title, "/shiporder/ships/ship/item/@title")]
```

2. The second datatype property bridge dpb10 has the XPath expression

   $xpath_{dpb_{10}}$ = `"/shiporder/items/item/title/text()"`

   and belongs to the concept bridge cb6 whose XPath is

   $xpath_{cb_6}$ = `"/shiporder/items/item"`

   So, we associate the variable of the subject node `?item` with the XPath $xpath_{cb_6}$ to make the start point:

   $vx_{from}$ `= (?item, "/shiporder/items/item")`

   Then, we associate the variable of the object node ?title with the XPath $xpath_{dpb_{10}}$ to make the end point:

   $vx_{to}$ `= (?title, "/shiporder/items/item/title/text()")`

   Thus, the second mapping pair goes from the start point $vx_{from}$ to the end point $vx_{to}$:

```
pair [from=(?item , "/shiporder/items/item"),
        to=(?title, "/shiporder/items/item/title/text()")]
```

Figure 11.2 shows those two pairs.

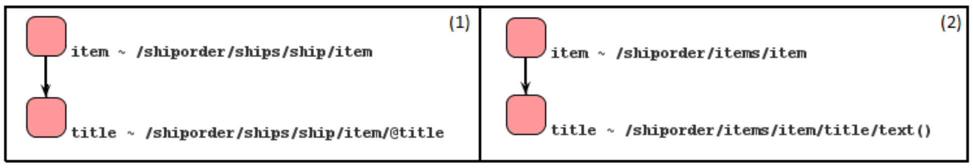

(1)
item ~ /shiporder/ships/ship/item
title ~ /shiporder/ships/ship/item/@title
(2)
item ~ /shiporder/items/item
title ~ /shiporder/items/item/title/text()

FIGURE 11.2: Two mapping pairs

### Case 2

If the predicate node is an object property *op*, we get its object property bridges *OPBs* from the XOML parser. For each object property bridge $opb \in OPBs$, we will generate a mapping pair as follows:

- We get the domain concept bridge $cb^{dom}$ of *opb* (to which it belongs), then we get the XPath expression $xpath_{cb^{dom}}$ of $cb^{dom}$. This XPath expression is associated with the variable of the subject node $v_{sub}$. This association $vx_{from} = (v_{sub}, xpath_{cb^{dom}})$ will be the start point of the target mapping pair.
- We get the range concept bridge $cb^{rng}$ of *opb* (to which it refers), then we get the XPath expression $xpath_{cb^{rng}}$ of $cb^{rng}$. This XPath expression is associated with the variable of the object node $v_{obj}$. This association $vx_{to} = (v_{obj}, xpath_{cb^{rng}})$ will be the end point of the target mapping pair.

For example, in the following triple pattern `{?ship ex:ship-item ?item}`, the predicate node is an object property `ex:ship-item`. This property has one object property bridge (thus we generate only one mapping pair for this bridge):

```
opb3 = OPB(ex:ship-item, cb3, cb4)
```

This object property bridge belongs to the (domain) concept bridge `cb3` whose XPath is:

$$xpath_{cb_3} = \texttt{"/shiporder/ships/ship"}$$

and refers to the (range) concept bridge `cb4` whose XPath is:

$$xpath_{cb_4} = \texttt{"/shiporder/ships/ship/item"}$$

So, we associate the variable of the subject node `?ship` with the XPath $xpath_{cb_3}$ to make the start point:

$$vx_{from} \texttt{ = (?ship, "/shiporder/ships/ship")}$$

Then, we associate the variable of the object node `?item` with the XPath $xpath_{cb_4}$ to make the end point:

$$vx_{to} \texttt{ = (?item, "/shiporder/ships/ship/item")}$$

Thus, the mapping pair goes from the start point $vx_{from}$ to the end point $vx_{to}$:

```
pair [from=(?ship, "/shiporder/ships/ship"),
        to=(?item, "/shiporder/ships/ship/item")]
```

The FIND-PAIRS algorithm is listed in Appendix L, Section L.1.

#### 11.2.4.2 Finding Mapping Graphs

Once mapping pairs are found for each triple pattern of the SPARQL query, we use them to establish possible mapping graphs.

A mapping graph is composed of a set of mapping pairs, such that for each triple pattern of the query, one of its mapping pairs is taken. Thus, the number of all possible mapping graphs equals the product of the sizes of mapping pairs sets for all triple patterns:

$$|MG| = \prod_{t \in T} |P^t|$$

The FIND-MAPPING-GRAPHS algorithm is listed in Appendix L, Section L.2.

Figure 11.3 shows mapping pairs generated for each triple pattern of our running example query. For the first triple pattern, there is one mapping pair $mp_1^1$, while for the second triple pattern, there is one mapping pair $mp_2^1$. For the third triple pattern, there are two mapping pairs $mp_3^1$, $mp_3^2$. Thus, two mapping graph are generated:

- the first (Figure 11.4-1) contains $mp_1^1$, $mp_2^1$, and $mp_3^1$, and
- the second (Figure 11.4-2) contains $mp_1^1$, $mp_2^1$, and $mp_3^2$.

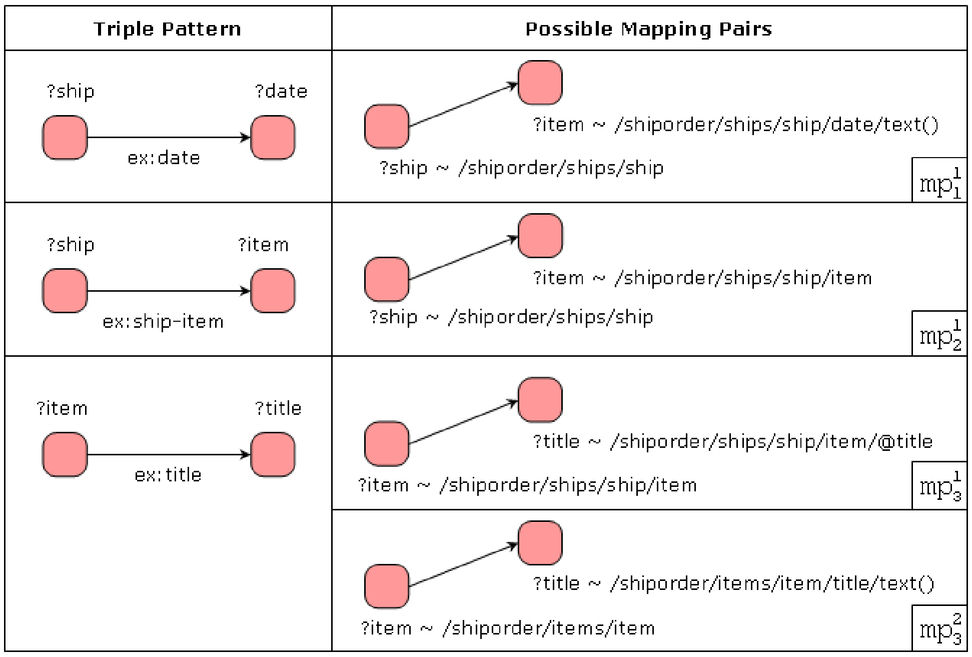

Triple Pattern
Possible Mapping Pairs
?ship
?date
ex:date
?item ~ /shiporder/ships/ship/date/text()
?ship ~ /shiporder/ships/ship
$mp_1^1$
?ship
?item
ex:ship-item
?item ~ /shiporder/ships/ship/item
?ship ~ /shiporder/ships/ship
$mp_2^1$
?item
?title
ex:title
?title ~ /shiporder/ships/ship/item/@title
?item ~ /shiporder/ships/ship/item
$mp_3^1$
?title ~ /shiporder/items/item/title/text()
?item ~ /shiporder/items/item
$mp_3^2$

FIGURE 11.3: Mapping Pairs for triple patterns of the running example query

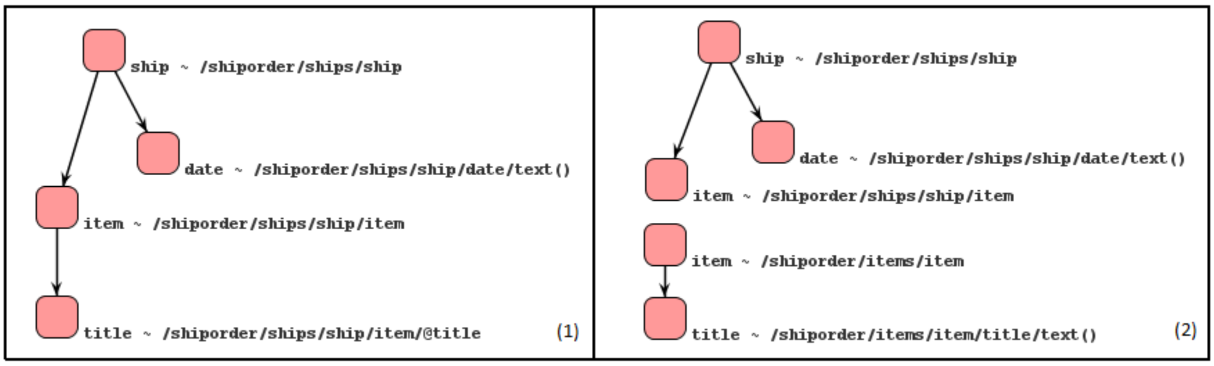

ship ~ /shiporder/ships/ship
date ~ /shiporder/ships/ship/date/text()
item ~ /shiporder/ships/ship/item
title ~ /shiporder/ships/ship/item/@title
(1)
ship ~ /shiporder/ships/ship
date ~ /shiporder/ships/ship/date/text()
item ~ /shiporder/ships/ship/item
item ~ /shiporder/items/item
title ~ /shiporder/items/item/title/text()
(2)

FIGURE 11.4: Two possible Mapping Graphs for the running example query

#### 11.2.4.3 Choosing Suitable Mapping Graph

Among the possible mapping graphs, we choose the suitable one to translate to XQuery. A suitable mapping graph is, simply, the one that has consistent mappings. That is, each variable is associated with one XPath expression only. Therefore, the suitable mapping graph is the connected graph. For example, in our running example, the first mapping graph (Figure 11.4-1) is connected, whereas the second one (Figure 11.4-2) is disconnected. Therefore, we choose the first mapping graph as the suitable mapping graph for translation to XQuery.

Recall that in an undirected graph $G$, two vertices $u$ and $v$ are called connected if $G$ contains a path from $u$ to $v$. Otherwise, they are called disconnected. A graph is called connected if every pair of distinct vertices in the graph can be connected through some path. A directed graph is called (weakly) connected if replacing all of its directed edges with undirected edges produces a connected (undirected) graph [1].

[1] http://en.wikipedia.org/wiki/Connectivity_(graph_theory)

| | A: adj. matrix of $G$ | B: adj. matrix of $\bar{G}$ | C: adj. matrix of $C(\bar{G})$ |
|---|---|---|---|
| MG1 | 0 0 0 0 | 0 0 0 1 | 1 1 1 1 |
| | 0 0 1 1 | 0 0 1 1 | 1 1 1 1 |
| | 0 0 0 0 | 0 1 0 0 | 1 1 1 1 |
| | 1 0 0 0 | 1 1 0 0 | 1 1 1 1 |
| MG2 | 0 0 1 0 0 | 0 0 1 0 0 | 1 0 1 0 0 |
| | 0 0 0 0 0 | 0 0 0 0 1 | 0 1 0 1 1 |
| | 0 0 0 0 0 | 1 0 0 0 0 | 1 0 1 0 0 |
| | 0 0 0 0 0 | 0 0 0 0 1 | 0 1 0 1 1 |
| | 0 1 0 1 0 | 0 1 0 1 0 | 0 1 0 1 1 |

TABLE 11.1: Adjacency matrices for mapping graphs, thier undirected versions and transitive closures

In order to determine wither a graph $G$ is connected, we follow the following procedure:

1. Firstly, we compute an undirected version of $G$ : $\bar{G}$
2. Then, we compute the transitive closure of $\bar{G}$ : $C(\bar{G})$
3. Finally, we say that the graph $G$ is connected if (and only if) $C(\bar{G})$ is complete.

For simplification, these operations are performed using the adjacency matrix[2].

In our running example, we have two possible mapping graphs shown in Figure 11.4. The adjacency matrices for these graphs, along with their undirected version and its transitive closure, are shown in Table 11.1.

When the transitive closure of the undirected version of a mapping graph is complete, then this mapping graph is connected. We can see that the first mapping graph satisfies this condition. Thus, the first mapping graph (Figure 11.5) is chosen for translation to XQuery as we will see in the next section.

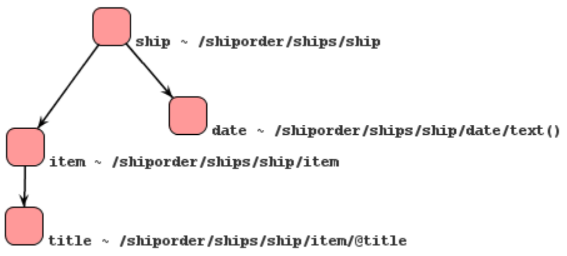

ship ~ /shiporder/ships/ship
date ~ /shiporder/ships/ship/date/text()
item ~ /shiporder/ships/ship/item
title ~ /shiporder/ships/ship/item/@title

FIGURE 11.5: Suitable Mapping Graph of the running example query

## 11.2.5 Building Target XQuery

This is the main step of query translation process. In this step, the different clauses of the XQuery query (`for`, `let`, `where`, `order by`, and `return`) are constructed using the information provided in the previous steps.

Since XQuery allows to return query results as new XML document, and since SPARQL results can also be formated as an XML document (see Appendix F), we design the target XQuery such that its returned results are formated using SPARQL Query Results XML Format. Hence, we

[2]http://en.wikipedia.org/wiki/Adjacency_matrix

don't need any further reformulation step as we did for the case of SPARQL-to-SQL translation in Section 8.5.

The target XQuery has the following pattern:

```
<sparql>
    <head>
        <variable name="..." />
        ...
    </head>
    <results>
    {
    for      ...
    let      ...
    where    ...
    order by ...
    return
        <result>
             <binding name="...">{ ... }</binding>
             ...
        </result>
    }
    </results>
</sparql>
```

The query starts with some element constructors that construct the head section of the SPARQL results document. Then, the body block of the query is a FLWOR expression that goes inside the results section of the SPARQL results document.

Let us now describe in details how to fill up this query pattern. We will need five sets:

1. a set of the selected variables of the original SPARQL query: $SV = \{\sigma_1, \sigma_2, \cdots, \sigma_k\}$.
2. a set of `for` statements $\mathbb{F} = \{\varphi_1, \varphi_2, \cdots, \varphi_l\}$, where each item $\varphi_i$ consists of a variable $\varphi_i^{var}$ and an XPath expression $\varphi_i^{xpath}$.
3. a set of `let` statements $\mathbb{L} = \{\lambda_1, \lambda_2, \cdots, \lambda_m\}$, where each item $\lambda_j$ consists of a variable $\lambda_j^{var}$ and an XPath expression $\lambda_j^{xpath}$.
4. a set of `order by` variables of the original SPARQL query: $OV = \{\theta_1, \theta_2, \cdots, \theta_n\}$
5. a set of `where` expressions: $\mathbb{W} = \{\omega_1, \omega_2, \cdots, \omega_p\}$

The set of the selected variables $SV$, and the set of order by variables $OV$, of the original SPARQL query, are already obtained during the SPARQL query parsing step (Section 12.4.4). Therefore, we present how to obtain the reminder of needed sets: $\mathbb{F}$, $\mathbb{L}$ and $\mathbb{W}$. This is mainly done by traversing the mapping graph obtained in the preprocessing step (Section 12.4.5), and by translating FILTER clauses of the original SPARQL query (obtained during the SPARQL query parsing step, Section 12.4.4).

### 11.2.5.1 Traversing the Mapping Graph

In the preprocessing step, we obtained a mapping graph that simulates the graph pattern of the SPARQL query. In this graph nodes are either variable-to-XPath associations or literal-to-XPath associations. In order to construct $\mathbb{F}$, $\mathbb{L}$ and $\mathbb{W}$ sets, the mapping graph is traversed depth-first:

If the visited vertex is a literal-to-XPath association `("value", xpath)`, then a where item is created: `xpath = "value"`, and added to $\mathbb{W}$ set.

Otherwise, if the visited vertex is a variable-to-XPath expression `(?v, xpath)`, then a `for` or a `let` statement is created according to the nature of the vertex being visited:

- If the vertex is non-leaf (internal node), then, a for statement is created, and added to $\mathbb{F}$ set: `for $v in xpath`
- If the vertex is leaf (external node), then, a let statement is created, and added to $\mathbb{L}$ set: `let $v := xpath`

In addition, absolute XPath expressions should be abbreviated into relative XPath expressions whenever possible. This is done by replacing some part of the XPath with the variable associated with its parent node. XPath expressions of root vertices remain absolute.

#### 11.2.5.2 Translation of FILTER Clauses

SPARQL FILTER clauses are translated to equivalent XQuery Where expressions. This translation is straightforward:

- Literal values are not changed.
- Comparing operators `(=, <, <=, ... )` are not changed (they exist in SPARQL and XQuery).
- SPARQL logical operators `(&&, ||, !)` are replaced by equivalent XQuery logical operators (and, or, not).
- Variable names remain the same, but the prefix `"?"` of SPARQL variables is replaced by the prefix `"$"` of XQuery variables.
- The SPARQL function `regex` is replaced by the XQuery function `fn:matches`.

For example, the SPARQL FILTER clause:

```
(regex(?x,"A","i") || !(?y>20))
```

is translated to the XQuery Where expression:

```
(fn:matches($x,"A","i") or not($y>20))
```

The where clauses translated from FILTER clauses are added to $\mathbb{W}$ set.

#### 11.2.5.3 Construction of the Target XQuery

When all the needed five sets are obtained, the target XQuery can be constructed using them. Considering that these sets are:

1. $SV = \{\sigma_1, \sigma_2, \cdots, \sigma_k\}$.
2. $\mathbb{F} = \{\varphi_1, \varphi_2, \cdots, \varphi_l\}$, where $\varphi_i = (\varphi_i^{var}, \varphi_i^{xpath})$.
3. $\mathbb{L} = \{\lambda_1, \lambda_2, \cdots, \lambda_m\}$, where $\lambda_j = (\lambda_j^{var}, \lambda_j^{xpath})$.
4. $OV = \{\theta_1, \theta_2, \cdots, \theta_n\}$.

```
<sparql>
   <head>
      <variable name="$\sigma_1$" />
      <variable name="$\sigma_2$" />
      ...
   </head>
   <results>
   {
   for $\varphi_1^{var}$ in $\varphi_1^{xpath}$
   for $\varphi_2^{var}$ in $\varphi_2^{xpath}$
   ...
   let $\lambda_1^{var}$ := $\lambda_1^{xpath}$
   let $\lambda_2^{var}$ := $\lambda_2^{xpath}$
   ...
   where $\omega_1$ and $\omega_2$ and ...  and $\omega_p$
   order by $\theta_1$, $\theta_2$, ..., $\theta_n$
   return
     <result>
        <binding name="$\sigma_1$">{ $\$\sigma_1$}</binding>
        <binding name="$\sigma_2$">{ $\$\sigma_2$}</binding>
        ...
     </result>
   }
   </results>
</sparql>
```

FIGURE 11.6: XQuery query pattern filled up

5. $\mathbb{W} = \{\omega_1, \omega_2, \cdots, \omega_p\}$.

The target XQuery is constructed by filling the query pattern up with the items of these sets as shown in Figure 11.6:

**Example** – Let us consider our running example SPARQL query. The selected variables of this query is: `SV = {?date, ?title}`, whereas the set of order by variables is empty `OV={}`.

We saw in Section 11.2.4.3, that the suitable chosen mapping graph of this example query is shown in Figure 11.5. In this graph, the non-leaf vertices are those of the variables `?ship` and `?item`, thus two `for` clauses are created for them:

```
for $ship in /shiporder/ships/ship
for $item in /shiporder/ships/ship/item
```

The leaf vertices are those of the variables `?date` and `?title`, thus two `let` clauses are created for them:

```
let $date := /shiporder/ships/ship/date/text()
let $title := /shiporder/ships/ship/item/@title
```

Then, absolute XPath expressions are abbreviated as follows:

The XPath of the root node `?ship` remains absolute: `/shiporder/ships/ship`

The children vertices of `?ship` are `?date` and `?item`, that have the XPath expressions:

`/shiporder/ships/ship/date/text()` and `/shiporder/ships/ship/item` respectively. Those expressions are abbreviated by replacing the `/shiporder/ships/ship` part by `$ship`:

`/shiporder/ships/ship/date/text()` ⟶ `$ship/date/text()`

`/shiporder/ships/ship/item` ⟶ `$ship/item`

the vertex `?item` has a child `?date` whose XPath is `/shiporder/ships/ship/item/@title`. This XPath is abbreviated by replacing the part `/shiporder/ships/ship/item` by `$item`:

`/shiporder/ships/ship/item/@title` ⟶ `$item/@title`

Thus, the `for` and `let` clauses become:

```
  for $ship in /shiporder/ships/ship
  let $date := $ship/date/text()
  for $item in $ship/item
  let $title := $item/@title
```

In the return clause, if a variable represents an attribute (so its XPath contains `"@"` symbol) then it is surrounded by the XQuery function `fn:data` in order to obtain the value of the attribute (otherwise, we will obtain: `@attribute = "value"`).

The target XQuery of our running example is:

```
<sparql>
  <head>
    <variable name="date" />
    <variable name="title" />
  </head>
  <results>
  {
  for $ship in /shiporder/ships/ship
  let $date := $ship/date/text()
  for $item in $ship/item
  let $title := $item/@title
  return
  <result>
    <binding name="date">{$date}</binding>
    <binding name="title">{fn:data($title)}</binding>
  </result>
  }
  </results>
</sparql>
```

The result of this query, formated as a SPARQL query results XML document, is:

```
<sparql>
   <head>
      <variable name="date"/>
      <variable name="title"/>
   </head>
   <results>
      <result>
         <binding name="date">12-01-2009</binding>
         <binding name="title">Empire Burlesque</binding>
      </result>
      <result>
         <binding name="date">12-01-2009</binding>
         <binding name="title">Hide your heart</binding>
      </result>
   </results>
</sparql>
```

## 11.3 Implementation

We have implemented a prototype of our SPARQL-to-XQuery query translation method. This prototypes is written in Java language and based on several freely-available APIs, namely:

- JENA – we use this API for read, manipulate and write SPARQL queries.
- JUNG – we use this API to create, manipulate and visualize the mapping graphs.
- Nux[3] – an open-source Java toolkit for efficient and powerful XML processing. We use this API for the execution of the translated XQuery query.

Figure 11.7 shows the exploitation of these APIs within the implementation.

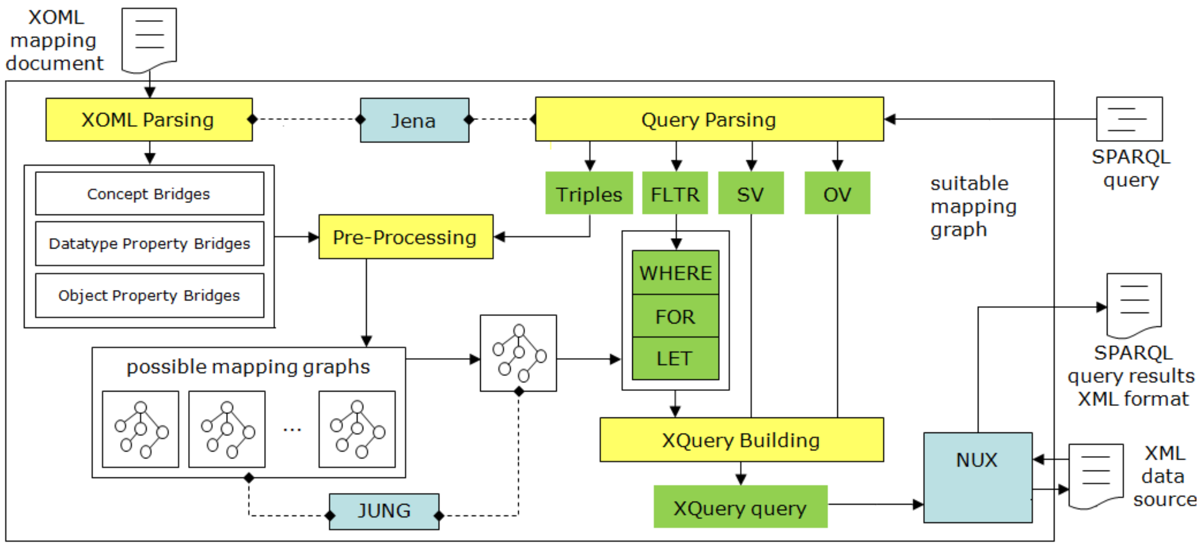


FIGURE 11.7: X2OWL implementation

## Chapter Summary

In this chapter, we have proposed a SPARQL-to-XQuery translation method. This translation process represents the second task of X2OWL, our tool for XML-to-Ontology (Chapter 10).

This translation method consists of four steps:

1. Parsing the XOML mapping document (Section 11.2.2).
2. Parsing the SPARQL Query (Section 11.2.3).
3. Preprocessing (Section 11.2.4).
4. Building the XQuery query (Section 11.2.5).

In Section 11.3, we discussed the implementation of the translation method.

[3] http://acs.lbl.gov/nux/

# Conclusion

This dissertation treated the area of ontology-based cooperation of information systems. We proposed a global architecture called OWSCIS that is based on ontologies and web-services for the cooperation of distributed heterogeneous information sources.

OWSCIS architecture uses local ontologies to represent the semantics of information sources (relational databases and XML data sources). In addition, OWSCIS uses a global ontology as a pivot global model for all participating local ontologies.

In this dissertation, we focused on the problem of connecting the local information sources to the local ontologies within OWSCIS architecture. This problem is articulated by three main axes:

- the creation of the local ontology from the local information sources,
- the mapping of local information sources to an existing local ontology, and
- the translation of queries over the local ontologies into queries over local information sources.

The problem of connecting the local information sources to the local ontologies depends on the type of information sources in consideration. Particularly, we distinguished relational databases and XML data sources. Considering the three axes mentioned above with these two types of information sources cases, we obtain six problems to solve. In this dissertation, we have proposed solutions for most of them.

For the first case (relational databases), we proposed a tool called DB2OWL that handles the three axes of database-to-ontology mapping:

1. ontology creation from a single database (Chapter 7 - Section 7.3),
2. mapping a database to an existing ontology (Chapter 7 - Section 7.4), and
3. SPARQL-to-SQL query translation (Chapter 8).

A prototype of this tool is implemented in Java programming language.

For the second case (XML data sources), we proposed a tool called X2OWL that handles two of the three axes of XML-to-ontology mapping:

1. ontology creation from a single XML data source (Chapter 10),
2. SPARQL-to-XQuery query translation (Chapter 11).

A prototype of X2OWL is also implemented in Java.

Thus, the axe of mapping an XML data source to an existing ontology is not treated in this dissertation. However, we think that this axe can be easily designed and implemented similarly to DB2OWL tool, since the XML-to-ontology mapping specification XOML has been already proposed.

## Future Works

The major future work is to make use of OWSCIS architecture. In fact, the techniques, methods and tools conducted within OWSCIS architecture need to be tested in case studies. The purpose is to evaluate all these techniques, methods and tools and to practice them with real data. This may lead to develop a fully comprehensive and functional system that put them together. Putting these techniques in use is necessary to evaluate the efficiency and scalability of these techniques with large information sources.

Some minor future works also include:

- **Developing OWSCIS modules as web-services** – In Chapter 3, we presented the global architecture of OWSCIS system. We said that this system is intended to be Web-Services based, that is, it consists of several modules and web services, each of them aims at performing a specific task, such as mapping web service, querying web service, and visualization web service. However, currently these parts have been developed as standard application prototypes. One of the future works is to develop these modules as web services.
- **Developing DB2OWL to support the mapping of databases to existing ontologies using DOML** – In Section 7.4, we described how DB2OWL allows users to map a database to an existing ontology. In the current version of DB2OWL, the graphical interface allows the user to provide mappings in the form of "Associations with SQL statements". In the future, we seek to develop another version of DB2OWL that allows also to specify mappings using DOML language.
- **Handling OPTIONAL clauses in the proposed query translation methods** – In chapters 8 and 11, we proposed SPARQL-to-SQL and SPARQL-to-XQuery query translation methods. These methods do not take into account OPTIONAL clauses. One of the future works is to improve our proposed translation methods such that they suitably handle OPTIONAL clauses.
- **Implementing our proposed approach for ontology building from multiple information sources** – we have proposed an ontology-evolution-based approach for ontology building from multiple information sources (databases and/or XML data sources). This work is presented in our paper: *Building Ontologies from Multiple Information Sources*, IT2009 [73], but is not presented in this dissertation due to lack of space. However, our future works include the implementation of this approach, and the proposition of a method for query processing over the ontology built from multiple information sources. Hence, besides query translation, the intended method should include a process of query decomposition over the multiple involved information sources.

# Appendices

# Appendix A

# Ontologies

## A.1 What is an Ontology?

The word ontology was taken from Philosophy, where it means a systematic explanation of being. In the 1990s, this word has been adopted by several artificial intelligence research communities. The notion of ontologies became widespread in fields such as intelligent information integration and retrieval on the internet, and knowledge management [153]. It is commonly understood that an ontology specifies a domain theory. Ontologies typically define concepts and their relations, together with constraints on those objects and relations.

One of the first definitions was given by Neches et al. [123], who defined an ontology as follows:

**Definition A.1.** An **ontology** defines the basic terms and relations comprising the vocabulary of a topic area as well as the rules for combining terms and relations to define extensions to the vocabulary.

This descriptive definition tells us that an ontology identifies basic terms and relations between terms, identifies rules to combine terms, and provides the definitions of such terms and relations. According to this definition, an ontology includes not only the terms that are explicitly defined in it, but also the knowledge that can be inferred from it.

Gruber [83] defined an ontology as follows:

**Definition A.2.** An **ontology** is an explicit specification of a conceptualization.

This definition became the most quoted in literature and by the ontology community. Borst [29] modified slightly Gruber's definition as follows:

**Definition A.3. Ontologies** are defined as a formal specification of a shared conceptualization.

A conceptualization refers to an abstract model of some phenomenon in the world which identifies relevant concepts of that phenomenon. Explicit means that the types of concepts used, and the constraints on their use, are explicitly defined. Formal refers to the fact that the ontology should be machine-readable. Shared means that the ontology arises from a consensus between several parties.

According to this definition, a database schema could be seen as an ontology. It is an abstract representation of a real-world phenomenon. It is also explicit and machine-readable. However, one could argue whether it represents a shared knowledge, that is, a common understanding of the domain between the parties involved. Database schemas are typically developed for one or a limited set of applications, whereas an ontology has to be agreed by several partners.

Guarino and Giaretta [85] provide the following definition:

**Definition A.4.** An **ontology** is a set of logical axioms designed to account for the intended meaning of a vocabulary.

Central to this definition are the concepts that are defined by a logical characterization through axioms.

Sometimes taxonomies are considered full ontologies [153]. The ontology community distinguishes ontologies that are mainly taxonomies from ontologies that model the domain in a deeper way and provide more restrictions on domain semantics. The community calls them *lightweight* and *heavyweight* ontologies respectively. On the one hand, lightweight ontologies include concepts, concept taxonomies, relationships between concepts, and properties that describe concepts. On the other hand, heavyweight ontologies add axioms and constraints to lightweight ontologies. Axioms and constraints clarify the intended meaning of the terms gathered on the ontology.

In general terms, an ontology is an organization of a set of terms related to a body of knowledge. Unlike a dictionary, which takes terms and provides definitions for them, an ontology organizes a knowledge on a basis of concepts. An ontology expresses a concept in a precise manner that can be interpreted and used by computer systems, whereas dictionary definitions are not processable by computer systems. Another difference is that by relying on concepts and specifying them precisely and formally we get definitive meanings that are independent of language or terminology.

## A.2 Ontology Types

There are many different types of ontologies, built for many different areas of applications. Lassila and McGuinness [109] classified different types of lightweight and heavyweight ontologies in a continuous line (also known as the *ontology spectrum*) according to their expressiveness (Figure A.1). The main categories and their meanings are:

- *Controlled vocabularies*, a finite list of terms. Catalogs are an example of this category.
- *Glossaries*, a list of terms with their meanings specified as natural language statements.
- *Thesauri*, that provide some additional semantics between terms. They give information such as synonym relationships, but do not supply an explicit hierarchy.
- *Informal **is-a** hierarchies*, there is an explicit hierarchy (generalization and specialization are supported), but there is no strict inheritance, i.e., an instance of a subclass is not necessarily also an instance of the superclass;
- *Formal **is-a** hierarchies*, there is a strict inheritance, i.e., if `B` is a subclass of `A` and an object is an instance of `B`, then the object is necessarily an instance of `A` as well. Strict subclass hierarchies are necessary to exploit inheritance.
- *Formal **is-a** hierarchies that include instances of the domain.*
- *Frame.* The ontology includes classes and their properties, which can be inherited by classes of the lower levels of the formal **is-a** taxonomy.
- *Ontologies that express value restriction.* These are ontologies that may place restrictions (for example, by a datatype) on the values that can fill a property.
- *Ontologies that express general logical constraints.* Values may be constraint by logical or mathematical formulas using values from other properties. Many ontologies allow some statement of disjointness of classes - i.e., it is impossible to be an instance of `A` and

simultaneously be an instance of B if A and B are disjoint classes. Some languages allow ontologists to state arbitrary logical statements. Very expressive ontology languages such as those seen in Ontolingua or CycL allow first order logic constraints between terms and more detailed relationships such as disjoint classes, disjoint coverings, inverse relationships, part-whole relationships, etc.

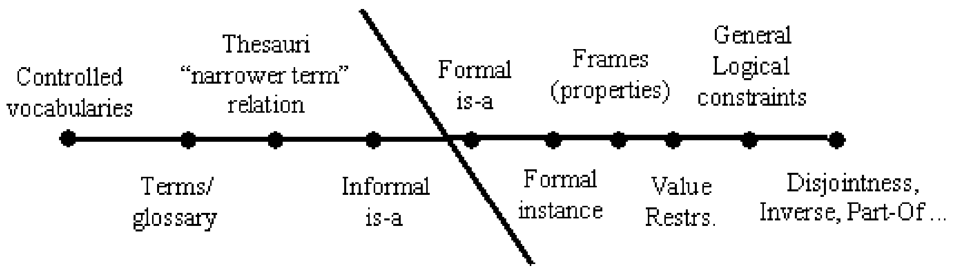


Figure A.1: Ontology Spectrum

The different types of ontologies identified in the literature can also be classified according to the subject of their conceptualization. Basically, we can distinguish domain ontologies and upper-level ontologies.

A domain ontology (or domain-specific ontology) models a specific domain, or part of the world (medical, pharmaceutical, engineering, law, enterprise, automobile, etc.). It provides vocabularies about concepts within a domain and their relationships, about the activities taking place in that domain, and about the theories and elementary principles governing that domain.

An upper-level ontology (also known as foundation ontology or top-level ontology) describes very general concepts that are common across the domains and give general notions under which all the terms in existing ontologies should be linked to. There is a clean boundary between domain and upper-level ontologies. The concepts in domain ontologies are usually specializations of concepts already defined in top-level ontologies, and the same might occur with the relations. Sometimes top-level ontologies are used to build domain ontologies, but often these are built first and then linked to upper-level ontologies. The main problem here is that there are several top-level ontologies, including SENSUS, GUM, MicroKosmos, SOWA, Cyc, and they differ on the criteria followed to classify the most general concepts of the taxonomy.

## A.3 Ontology Components

In this section we introduce the common components of an ontology, that is, the main kind of components used to describe domain knowledge in ontologies. Gruber [83] stated that knowledge in ontologies can be formalized using five kind of components: concepts, relations, functions, axioms and instances. Concepts in the ontology are usually organized in taxonomies. Other components like procedures and rules are also identified in some ontology languages (i.e., OCML).

## Concepts

Concepts (or classes) are used in a broad sense. They can represent abstract concepts (intentions, beliefs, feelings, etc.) or concrete concepts (people, computers, tables, etc.). They can be elementary (electron) or composite (atom), real or fictitious. In short, a concept can be anything about which something is said, and, therefore, could also be the description of a task, function, action, strategy, reasoning process, etc. Classes in the ontology are usually organized in taxonomies. Taxonomies are widely used to organize ontological knowledge in the domain using generalization/specialization relationship through which simple/multiple inheritance can be applied.

In some paradigms, metaclasses can also be defined. Metaclasses are classes whose instances are classes. They usually allow for gradations of meaning, since they establish different layers of classes in the ontology where they are defined.

## Relations

Relations represent a type of association between concepts of the domain. They are formally defined as any subset of a product of $n$ sets, that is: $R \subset C_1 \times C_2 \times \cdots \times C_n$ ($n$-ary relations). Ontologies usually contain binary relations that have two arguments. The first argument is known as the domain of the relation, and the second argument is the range. For instance, the binary relation *Subclass-Of* is used for building the class taxonomy. Binary relations are also used to connect different taxonomies. Examples of binary relations are *part-of* and *connected-to*. Relations can be instantiated with knowledge from the domain. Binary relations are sometimes used to express concept attributes (slots).

## Attributes

Attributes (or slots) are properties, features, characteristics, or parameters that objects and classes can have. Attributes are usually distinguished from relations because their range is a datatype, such as *string*, *number*, etc., while the range of relations is a concept.

## Functions

Functions are complex structures formed from certain relations that can be used in place of an individual term in a statement. They are defined as mappings between a list of input arguments and its output argument. Functions are a special case of relations in which the n-th element of the relation is unique for the n-1 preceding elements. This is formally expressed as: $F : C_1 \times C_2 \times \cdots \times C_{n-1} \rightarrow C_n$.

## Axioms

Axioms are assertions (including rules) in a logical form that together comprise the overall theory that the ontology describes in its domain of application. According to Gruber, formal axioms serve to model sentences that are always true. They are normally used to represent knowledge that cannot be formally defined by the other components. They can be included in an ontology for several purposes, such as constraining the information contained in the ontology, verifying the consistency (correctness) of the ontology itself or the consistency of the knowledge stored in a knowledge base, or deducting new information.

### Instances

Instances are used to represent elements or individuals in an ontology. They are the basic, "ground level" components of an ontology. The individuals in an ontology may include concrete objects such as people, animals, tables, automobiles, molecules, and planets, as well as abstract individuals such as numbers and words. Strictly speaking, an ontology need not include any individuals, but one of the general purposes of an ontology is to provide a means of classifying individuals, even if those individuals are not explicitly part of the ontology. *Facts* is the term commonly used to represent a relation which holds between instances.

Ontologies are commonly encoded using ontology languages. The next section is devoted to present some of most known ontology languages.

## A.4 Ontology Languages

In the literature, there are many ontology specification languages. We can distinguish between two main families of ontology languages:

- **Traditional ontology languages** – such as Ontolingua, OKBC, OCML, F-Logic, and LOOM.
- **Ontology markup languages** – such as XML, RDF(S), XOL, SHOE, OIL, DAML, DAML+OIL, and OWL.

These languages are based on different knowledge representation paradigms such as Description Logics (CLASSIC, OIL, LOOM, CARIN, AL-log, DLR, and OWL-DL), and Frame-based systems (F-Logic, OKBC, Ontolingua). We refer to [47] and [165] for a comparison of ontology languages.

### A.4.1 Traditional Ontology Languages

In the past years, a set of languages have been used for implementing ontologies: KIF, Ontolingua, LOOM, OKBC, OCML and F-Logic (Figure A.2). Knowledge representation (KR) paradigms underlying these languages are diverse: frame-based, description logic, first order predicate calculus, object-oriented, etc. However, OKBC is not an ontology language but a protocol that allows the access to KR systems using primitives based on frames.

#### KIF

KIF (Knowledge Interchange Format) [71] was developed in 1991 to solve the problem of language heterogeneity in knowledge representation, and to allow the interchange of knowledge between

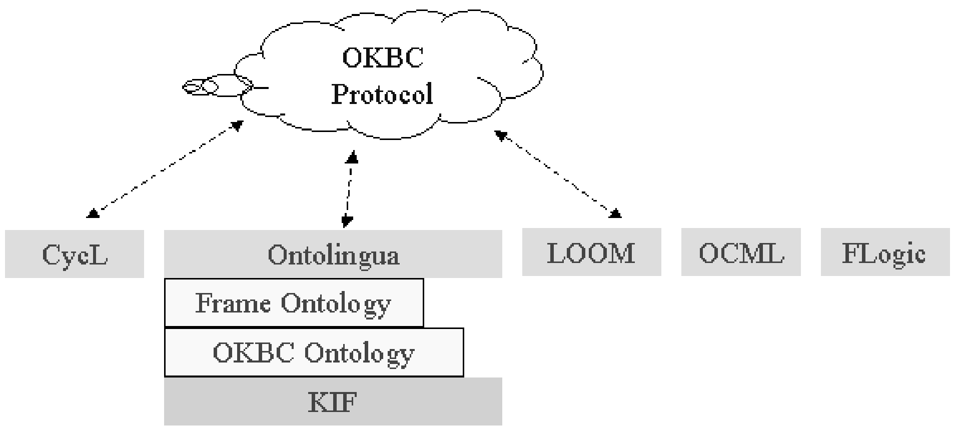


Figure A.2: Traditional ontology languages

diverse information systems. KIF is a prefix notation of first order predicate calculus with some extensions. It permits the definition of objects, functions and relations with functional terms and equality. KIF has declarative semantics (it is possible to understand the meaning of expressions without an interpreter to manipulate them). It also permits the representation of meta-knowledge, reifying functions and relations, and non-monotonic reasoning rules.

As KIF is an interchange format, it is a very tedious task to implement ontologies with it. But with the Frame Ontology [83], built on top of KIF, this task becomes easier. The Frame Ontology is a KR ontology for modeling ontologies under a frame-based approach and provides primitives such as Class, Binary-Relation, Named-Axiom, etc. Since it was built on the basis of KIF and a series of extensions of this language, this ontology can be completely translated into KIF with the Ontolingua Server's translators. In 1997, the Frame Ontology was modified because another representation ontology, the OKBC Ontology, was included between KIF and the Frame Ontology, as shown in Figure A.2. As a result, several definitions from the Frame Ontology were moved to the OKBC Ontology.

### OntoLingua

Ontolingua [82] was released in 1992 by the Knowledge Systems Laboratory of Stanford University. It is an ontology language based on KIF and on the Frame Ontology. Ontolingua is the ontology-building language used by the Ontolingua Server [60].

Both the Frame Ontology and the OKBC Ontology are less expressive than KIF. This means that not all the knowledge that can be expressed in KIF can be expressed only with the primitives provided by these KR ontologies. Thus Ontolingua allows adding KIF expressions to the definitions implemented with the Frame Ontology and the OKBC Ontology. With the Ontolingua language we can build ontologies according to any of the four following approaches: (1) using the Frame Ontology vocabulary; (2) using the OKBC Ontology vocabulary; (3) using KIF expressions; and (4) combining the Frame Ontology vocabulary, the OKBC Ontology vocabulary and KIF expressions simultaneously.

Ontolingua ontologies are kept at the Ontolingua Server, which besides storing Ontolingua ontologies it allows users to build ontologies using its ontology editor, to import ontologies implemented in Ontolingua, and to translate ontologies from Ontolingua into other languages.

### LOOM

LOOM [114][115] is a high-level programming language and environment intended for use in constructing general-purpose expert systems and other intelligent application programs. LOOM was being developed from 1986 until 1995 by the Information Science Institute (ISI) of Southern California University.

LOOM is a descendent of the KL-ONE [32] family of languages, characterized for their efficient automatic classifiers. LOOM achieves a tight integration between rule-based and frame-based paradigms. LOOM is based on the description logics (DL) paradigm and is composed of two different sublanguages: the “description” and the “assertion” languages. The former is used for describing domain models through objects and relations (TBox), the latter allows specifying constraints on concepts and relations, and asserting facts about individuals (ABox).

Procedural programming is supported through pattern-directed methods, while production-based and classification-based inference capabilities support a powerful deductive reasoning (in the form of an inference engine: the classifier). All of these capabilities reside in a framework of query-based assertion and retrieval.

### OKBC

OKBC [41] is the acronym for *Open Knowledge Base Connectivity*, previously known as the *Generic Frame Protocol* (GFP) [95]. OKBC is appeared in 1997 as the result of the joint efforts of the Artificial Intelligence Center of SRI International and the Knowledge Systems Laboratory of Stanford University. The objective of OKBC was to create a frame-based protocol to access knowledge bases stored in different knowledge representation systems (KRSs), as Figure A.2 shows. It is considered complementary to language specifications developed to support knowledge sharing.

The GFP Knowledge Model, which is the implicit representation formalism underlying OKBC, supports an object-centered representation of knowledge and provides a set of representational constructs commonly found in frame representation systems: constants, frames, slots, facets, classes, individuals and knowledge bases. OKBC also defines procedures (using a Lisp-like syntax) to describe complex operations that cannot be performed by simply using the protocol primitives. Different implementations of the OKBC protocol have been provided for different KRSs.

### OCML

OCML [122] stands for *Operational Conceptual Modeling Language*. It was developed in 1998 at the Knowledge Media Institute (UK) in the context of the VITAL project [142] to provide operational modeling capabilities for the VITAL workbench [57].

Several pragmatic considerations were taken into account in the development of OCML. One of them is the compatibility with standards, such as Ontolingua, so that OCML can be considered as a kind of "operational Ontolingua", providing theorem proving and function evaluation facilities for its constructs.

As OCML is mainly based on Ontolingua, it is a frame-based language with a Lisp-like syntax. Thus, OCML provides primitives to define classes, relations, functions, axioms and instances. It also provides primitives to define rules (with backward and forward chaining). In order to make the execution of the language more efficient, it also adds some extra logical mechanisms for efficient reasoning, such as procedural attachments. A general tell&ask interface is also implemented, as a mechanism to assert facts and/or examine the contents of an OCML model.

OCML has a basic ontology library holding 12 ontologies. The first group of ontologies (*lists*, *numbers*, *sets* and *strings*) gives basic definitions for handling basic data types. The second (*frames*, *relations* and *functions*) deals with definitions for knowledge representation in OCML. The third (*task-method*, *mapping* and *inferences*) is used for building problem solving methods. Finally, the last group (*meta* and *environment*) describes the OCML language. For editing and browsing OCML ontologies, the WebOnto ontology editor [56] can be used.

### FLogic

FLogic [99] is an acronym for *Frame Logic*. It was created in 1995 at the Department of Computer Science of the State University of New York. FLogic was initially developed as an object oriented approach to first order logic. It was specially used for deductive and object-oriented databases, and was later adapted and used for implementing ontologies.

FLogic integrates features from object-oriented programming, frame-based KR languages and first-order predicate calculus. It accounts in a clean and declarative fashion for most of the structural aspects of object-oriented and frame-based languages. These features include object identity, complex objects, inheritance, polymorphic types, query methods, encapsulation, and others. In FLogic, the term *object* is used instead of *frame*, and *attribute* instead of *slot*.

FLogic stands in the same relationship to the object-oriented paradigm as classical predicate calculus stands to relational programming. FLogic has a model-theoretic semantics and a sound and complete resolution-based proof theory.

FLogic ontologies can be built with different ontology development tools such as OntoEdit, Protégé-2000 [125] and WebODE. Applications of this language range from object-oriented and deductive databases to ontologies. For instance, FLogic was used in the ontology-related project $(KA)^2$ [15], for information integration applications [112], etc.

### CycL

The CycL language has been developed in 1990 within the Cyc project [110]. CycL is a formal language whose syntax derives from first-order predicate calculus. In order to express real-world expertise and common sense knowledge, however, it goes far beyond first order logic. The vocabulary of CycL consists of terms: semantic constants, non-atomic terms (NATs), variables, numbers, strings, etc. Terms are combined into meaningful CycL expressions, ultimately forming meaningful closed CycL sentences (with no free variables.) A set of CycL sentences forms a knowledge base.

### A.4.2 Ontology Markup Languages

In the recent years, new web standard languages have been created such as XML [33] and RDF [104]. As a consequence, new XML-based ontology specification languages have also emerged: SHOE [89], XOL [94], OIL [62], as well as RDF Schema [35] and XML Schema [155][22]. The role of new languages in this scenario is twofold: they can be used to provide the semantics of information contained in electronic documents or can be used for the exchange of ontologies across the web.

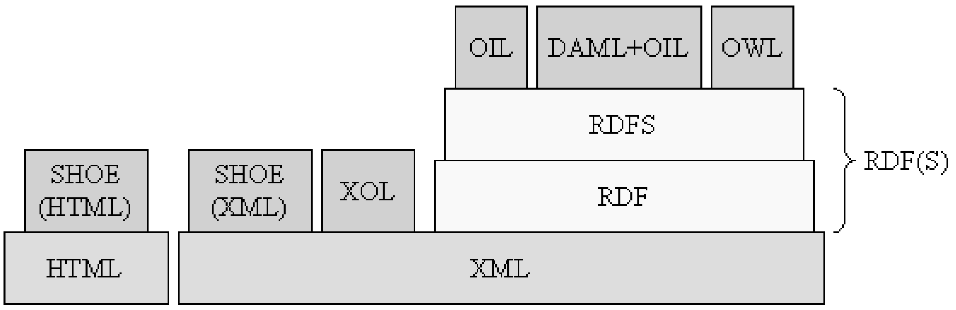


FIGURE A.3: Ontology markup languages

#### SHOE

SHOE [113][89] stands for Simple HTML Ontology Extension. It was developed in the University of Maryland (USA) in 1996. SHOE was first an extension of HTML by adding tags that are necessary to embed semantic data into web pages. This extension contains two categories: tags for constructing ontologies and tags for annotating web documents. Recently, SHOE has been adapted in order to be XML compliant.

The intent of this language is to make it possible for agents to gather meaningful information about web pages and documents, improving search mechanisms and knowledge-gathering. The two-phase process to achieve it consists of: (1) defining an ontology describing valid classifications of objects and valid relationship between them; (2) annotating HTML pages to describe themselves, other pages, etc.

SHOE allows representing concepts, their taxonomies, n-ary relations, instances and deduction rules, which are used by its inference engine to obtain new knowledge.

#### XML

XML [33] is a subset of the ISO standard SGML (Standard General Markup Language). SGML [92] was created to specify document markup languages or tag sets for describing electronic texts. XML development was started in 1996 by the XML Working Group of the World Wide Web Consortium (W3C), for ease of implementation and interoperability with both SGML and HTML. XML became a W3C Recommendation in February 1998.

XML was designed to overcome some of the drawbacks of HTML. For instance, HTML was meant to be consumed only by human readers since it dealt only with content presentation of Web resources, not with the structure of that content. Currently, XML is being used not only to structure texts but to exchange a wide variety of data on the Web, allowing better interoperability between information systems. XML allows users to define their own tags and attributes, define data structures (nesting them), extract data from documents and develop applications which test the structural validity of a XML document.

As a language for the World Wide Web, the main advantages of XML are: 1) it is easy to parse, 2) its syntax is well defined, and 3) it is human readable.

XML itself has no special features for the specification of ontologies, as it just offers a simple but powerful way to specify a syntax for an ontology specification language. Therefore, XML will be used for two purposes: for providing the syntax of a set of languages, such as XOL or OIL, so that the definition of these languages just consists of describing the semantics of new tags created and used in it; and for covering ontology exchange needs, exploiting the communication facilities of the World Wide Web.

When using XML as the basis for an ontology specification language (XML-based ontology languages), its main advantages are:

1. The definition of a common syntactic specification by means of a DTD (Document Type Definition).
2. Information coded in XML is easily readable for humans (although it is not intended to be used for the direct coding of ontologies, information of the ontology coded in an XML-based ontology language can be easily read and understood).
3. It can be used to represent distributed knowledge across several web-pages, as it can be embedded in them.

## XOL

XOL [94] stands for XML-Based Ontology Exchange Language. XOL was designed in 1999 to provide a format for exchanging ontology definitions among a set of interested parties. Therefore, it is not intended to be used for the development of ontologies, but as an intermediate language for transferring ontologies among different database systems, ontology-development tools or application programs.

XOL allows to define in a XML syntax a subset of OKBC, called OKBC-Lite. As OKBC defines a protocol for accessing frame-based representation systems, XOL may be suitable for exchanging information between different systems, via the WWW. The main handicap is that frames (defined in OKBC) are excluded from this language, and only classes (and their hierarchies), slots and facets can be defined. However, since XOL files are textual, a text editor or XML editor may be used to author XOL files. It is expected that many XML tools will soon be available so that XOL documents will be easily generated with them.

### RDF(S): RDF and RDF Schema

RDF [104] stands for Resource Description Framework. It is being developed by the W3C for the creation of metadata describing Web resources. It became a W3C Recommendation in February 2004.

The goal of RDF is to define a mechanism for describing resources that makes no assumptions about a particular application domain nor the structure of a document containing information. The definition of the mechanism should be domain neutral, and the mechanism should be suitable for describing information about any domain. RDF is an infrastructure that enables the encoding, exchange and reuse of structured metadata. Search engines, intelligent agents, information broker, browsers and human user can make use of semantic information.

RDF has been and will be applied in various areas. It provides better capabilities for search engine in resource discovery. It describes the content and content relationships in formalized way to provide computer-understandable catalogues. It facilitates the knowledge sharing and exchange among various intelligent software agents.

The data model of RDF consists of three object types:

**Resources** – They are entities that can be referred to by URIs. A resource may be an entire Web page; a part of a Web page; a whole collection of pages; or an object that is not directly accessible via the Web.

**Properties** – A property is a specific aspect, characteristic, attribute, or relation used to describe a resource.

**Statements** – An RDF statement consists of a specific resource together with a named property plus the value of that property for that resource. These three individual parts of a statement are called, respectively, the *subject*, the *predicate*, and the *object*. In a nutshell, RDF defines object-property-value triples as basic modeling primitives that provide a syntax-neutral way of representing RDF expressions. As RDF statements are also resources, statements can be recursively applied to statements allowing their nesting.

RDF properties may be thought of as attributes of resources and in this sense correspond to traditional attribute-value pairs. RDF properties also represent relationships between resources. As such, the RDF data model can therefore resemble an entity-relationship diagram. However, it doesn't provide mechanisms for declaring these properties, and the relationships between these properties and other resources. This is the role of RDFS, the acronym for RDF Schema Specification language [35], which is a declarative language used for the definition of RDF schemas.

It is based on some ideas from knowledge representation (semantic nets, frames and predicate logic), but it is much simpler to implement (and also less expressive) than full predicate calculus languages such as CycL and KIF. Core classes are *class*, *resource* and *property* (Figure A.4); hierarchies and type constraints can be defined (core properties are *type*, *subClassOf*, *subPropertyOf*, *seeAlso* and *isDefinedBy*). Some core constraints are also defined.

RDF(S) is the term commonly used to refer to the combination of RDF and RDFS. Thus, RDF(S) combines semantic networks with frames but it does not provide all the primitives that are usually found in frame-based knowledge representation systems. In fact, neither RDF, nor RDFS, and nor their combination in RDF(S) should be considered as ontology languages by itself, but rather as general languages for describing metadata in the Web.

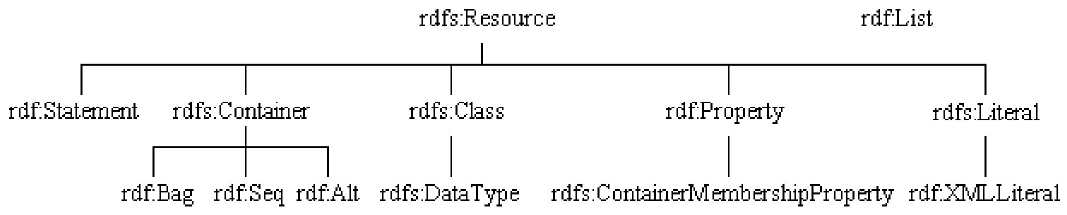


FIGURE A.4: Class taxonomy of RDF(S)

RDF(S) is widely used as a representation format in many tools and projects, and there exists a huge amount of resources for RDF(S) handling, such as browsing, editing, validating, querying, storing, etc.

A conclusion is that an ontology defined in RDF(S) will lack from functions and axioms, but concepts, relations and instances can be easily defined.

### OWL

Since OWL is the ontology language used in the reminder of this dissertation, we will give more focus on it. In the following section, we give an overview on the primitives provided by OWL and we introduce its three layers: OWL Lite, OWL DL and OWL Full.

Figures A.5 shows a timeline of ontology languages.

## A.5 OWL

The OWL language [118] has been created by the W3C Web Ontology (WebOnt) Working Group. It is derived from the DAML+OIL language, and it builds upon RDF(S). OWL is intended for publishing and sharing ontologies in the Web. OWL has become a W3C recommendation since February 2004.

Main features of OWL language include:

- **Ontologies** – An OWL ontology is a sequence of axioms and facts, plus inclusion references to other ontologies, which are considered to be included in the ontology. OWL ontologies are web documents, and can be referenced by means of a URI. Ontologies also have a non-logical component (not yet specified) that can be used to record authorship, and other non-logical information to be associated with a ontology.

- **Axioms** – Axioms are used to associate class and property IDs with either partial or complete specifications of their characteristics, and to give other logical information about classes and properties. It contains Class Axioms, Property axioms, Descriptions and Restrictions.

- **Facts** – Facts state information about particular individuals in the form of a class that the individual belongs to plus properties and values. Individuals can either be given an individual ID or be anonymous (blank nodes in RDF terms). The syntax here is set up to mirror the normal RDF/XML syntax.

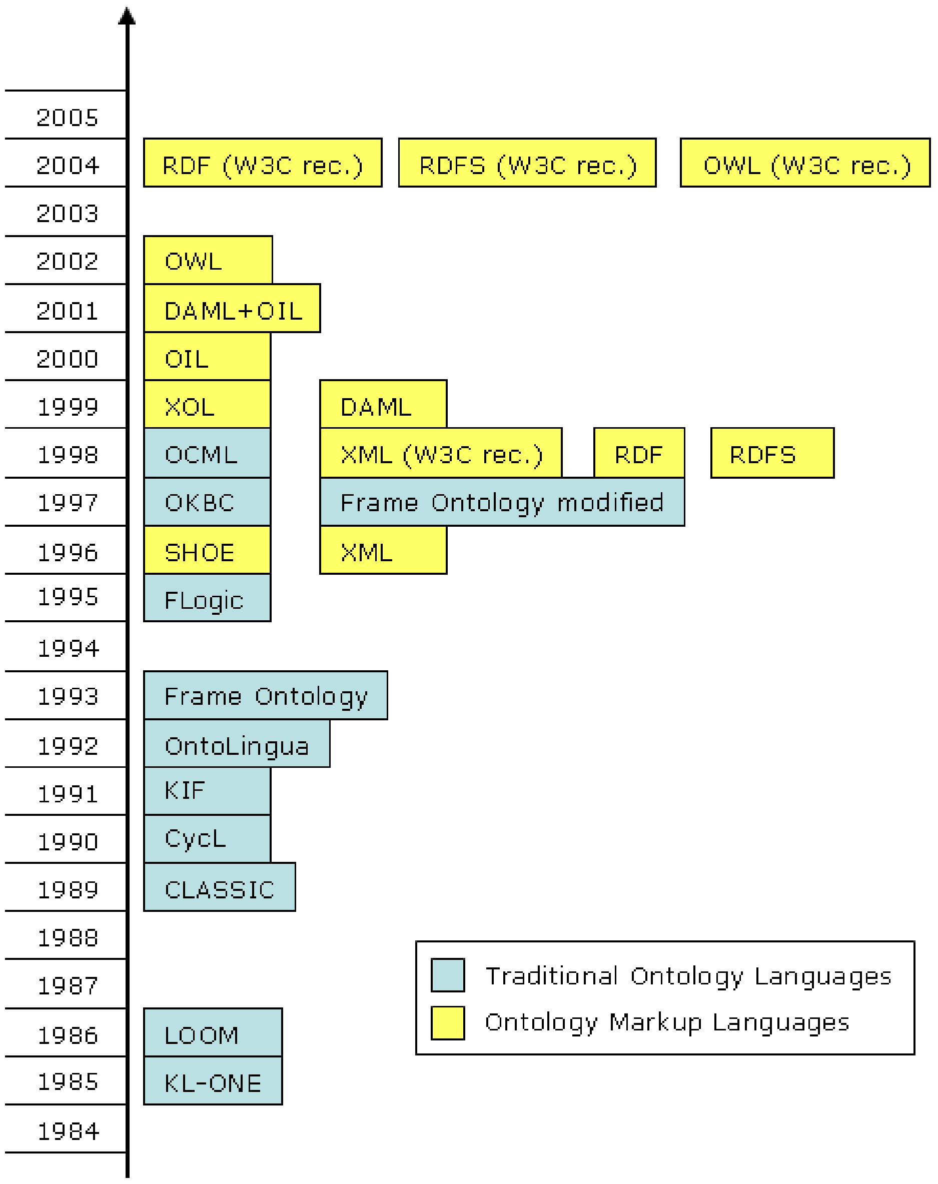


FIGURE A.5: History of ontology languages

The standard semantics of OWL are based on an *open world assumption* (OWA). This means that we cannot assume that all information is known about all the individuals in the domain. Thus, simply being unable to prove that an individual a is an instance of X does not justify our concluding that a is not an instance of X. This is in contrast to languages or systems using *negation as failure* or a *closed world assumption* (CWA). The OWA facilitates reasoning about intentional definitions of classes - we do not need to know all the information about the world in order to be able to make deductions about the relationships between classes.

### A.5.1 Concepts in OWL

Concepts are known as classes in OWL and are created with the primitive `owl:Class`. OWL classes can be constructed either:

1) Explicitly, using a class identifier:

```
<owl:Class rdf:ID="Lecturer" />
```

2) By exhaustively enumerating the individuals of the class using `owl:oneOf`:

```
<owl:Class rdf:ID="ComputingOfficer">
    <owl:oneOf rdf:parseType="Collection">
        <Academic rdf:about="#RGhawi" />
        <Academic rdf:about="#TPoulain" />
        <Academic rdf:about="#GGomez" />
    </owl:oneOf>
</owl:Class>
```

3) Using logical operators: `owl:intersectionOf`, `owl:unionOf`, `owl:complementOf`:

```
<owl:Class rdf:ID="Assistant">
    <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Student"/>
        <owl:Class rdf:about="#Lecturer"/>
    </owl:intersectionOf>
</owl:Class>
```

4) Using property restriction:

```
<owl:Class rdf:ID="Researcher">
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="#activity" />
            <owl:someValuesFrom rdf:resource="#ResearchArea" />
        </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>
```

In OWL, concept taxonomies can be created using `rdfs:subClassOf` for primitive concepts, and `owl:intersectionOf`, `owl:unionOf`, or `owl:equivalentClass` for defined concepts. Disjoint knowledge can be expressed with the `owl:disjointWith` primitive.

### A.5.2 Properties in OWL

In OWL there are two types of properties: `owl:ObjectProperty`, whose range is a class, and `owl:DatatypeProperty`, whose range is a datatype. To define a property we may explicit its domain and range with the primitives `rdfs:domain` and `rdfs:range`.

`owl:ObjectProperty` – serves to define properties that connect a class with another class (relations).

```
<owl:ObjectProperty rdf:ID="activity">
  <rdfs:domain rdf:resource="Person" />
  <rdfs:range rdf:resource="ActivityArea" />
</owl: ObjectProperty>
```

`owl:DatatypeProperty` – serves to define properties that connect a class with a datatype (attributes).

```
<owl:DatatypeProperty rdf:ID="name">
  <rdfs:domain rdf:resource="Person" />
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema/string" />
</owl:DatatypeProperty>
```

In addition, OWL language also provides the primitives `owl:TransitiveProperty` and `owl:SymmetricProperty` that serve to define logical characteristics of properties, and the primitives `owl:FunctionalProperty` and `owl:InverseFunctionalProperty` which are used to define global cardinality restrictions of properties. Finally, `owl:AnnotationProperty` is used to define properties that have no logical consequences on an OWL ontology, but just give information about its classes, properties, individuals or the whole ontology. All these seven kinds of properties specialize the class `rdf:Property` (they are rdfs:subClassOf it).

We can state that a property is a subproperty of other properties with the primitive `rdfs:subPropertyOf`. Finally, we can express equivalence between properties with `owl:equivalentProperty`, and inverse properties with `owl:inverseOf`.

### A.5.3 Property Constraints

Constraints define classes in terms of a restriction that they satisfy with respect to a given property. They are defined with the primitive `owl:Restriction`. Restrictions are defined with two elements: `owl:onProperty` (which refers to the property name) and another element that expresses either a value constraint (`owl:allValuesFrom`, `owl:someValuesFrom`, and `owl:hasValue`), or a cardinality constraint (`owl:cardinality`, `owl:maxCardinality`, and `owl:minCardinality`).

```
<owl:Restriction>
  <owl:onProperty rdf:resource="#hasFather"/>
  <owl:maxCardinality rdf:datatype="xsd:nonNegativeInteger">
    1
  </owl:maxCardinality>
</owl:Restriction>

<owl:Restriction>
      <owl:onProperty rdf:resource="#bakes"/>
      <owl:someValuesFrom rdf:resource="#Bread"/>
</owl:Restriction>
```

### A.5.4 OWL Layering

OWL is divided in three layers: OWL Lite, OWL DL, and OWL Full. Every legal OWL Lite ontology is a legal OWL DL ontology, and every legal OWL DL ontology is a legal OWL Full ontology.

**OWL Lite** – extends RDF(S) and gathers the most common features of OWL. It is mainly intended for class hierarchies and simple constraints features. For example, while OWL Lite supports cardinality constraints, it only permits cardinality values of 0 or 1.

**OWL DL** – is a sublanguage of OWL which includes all OWL language constructs, but places a number of constraints on the use of them. OWL DL is based on Description Logic theoretical properties, therefore, it is intended where completeness and decidability are an issue. The main features of OWL DL are:

- OWL DL requires a pairwise separation between classes, datatypes, datatype properties, object properties, annotation properties, ontology properties, individuals, data values and the built-in vocabulary (e.g. a class can not also be an individual or property, a property can not also be an individual or class).
- In OWL DL the set of object properties and datatype properties are disjoint. This implies that the following four property characteristics: inverse of, inverse functional, symmetric, and transitive can never be specified for datatype properties.
- OWL DL requires that no cardinality constraints (local nor global) can be placed on transitive properties or their inverses or any of their superproperties.
- Most RDF(S) vocabulary cannot be used within OWL DL.

**OWL Full** – is the complete OWL language. OWL Full allows free mixing of OWL with RDF Schema and does not enforce a strict separation of classes, properties, individuals and data values. OWL Full provides the maximum of expressivity and more flexibility to represent ontologies than OWL DL does. For example, a class can be treated simultaneously as a collection of individuals and as an individual in its own right. However, the use of OWL Full features means that one loses some guarantees that OWL DL and OWL Lite can provide for reasoning systems.

In OWL Full the resource `owl:Class` is equivalent to `rdfs:Class`. This is different from OWL DL and OWL Lite, where `owl:Class` is a proper subclass of `rdfs:Class`. OWL Full also allows classes to be treated as individuals. In OWL Full object properties and datatype properties are not disjoint. In OWL Full `owl:ObjectProperty` is equivalent to `rdf:Property`. The consequence is that datatype properties are effectively a subclass of object properties.

Table A.1 summaries the differences between the three "spices" of OWL.

Table A.1: Differences between OWL Lite, DL, and Full

| | **OWL Lite** | **OWL DL** | **OWL Full** |
|---|---|---|---|
| Compatibility with RDF | Theoretically, no RDF document can be assumed to be compatible with OWL Lite | Theoretically, no RDF document can be assumed to be compatible with OWL DL | All valid RDF documents are OWL full |
| Restrictions on class definition | Requires separation of classes, instances, properties, and data values | Requires separation of classes, instances, properties, and data values | Classes can be instances or properties at the same time |
| RDF Mixing | Restricts mixing of RDF and OWL constructs | Restricts mixing of RDF and OWL constructs | Freely allows mixing of RDF and OWL constructs |
| Classes Descriptions | The only class description available in OWL Lite is IntersectionOf | Classes can be UnionOf, ComplementOf, IntersectionOf, and enumeration | Classes can be UnionOf, ComplementOf, IntersectionOf, and enumeration |
| Cardinality Constraints | Cardinality: 0/1<br>MinCardinality: 0/1<br>MaxCardinality: 0/1 | Cardinality $\geq 0$<br>MaxCardinality $\geq 0$<br>MinCardinality $\geq 0$ | Cardinality $\geq 0$<br>MaxCardinality $\geq 0$<br>MinCardinality $\geq 0$ |
| Value Constraints | `owl:allValuesFrom`<br>`owl:someValuesFrom`.<br>Object of `owl:allValuesFrom` should be a class name | `owl:allValuesFrom`<br>`owl:someValuesFrom`<br>`owl:hasValue` | `owl:allValuesFrom`<br>`owl:someValuesFrom`<br>`owl:hasValue` |
| Metamodeling | Does not allow metamodeling | Does not allow metamodeling | Allows metamodeling. Thus RDF and OWL constructs can be augmented or redefined |
| Class | `owl:Class` is subclass of `rdfs:Class` | `owl:Class` is subclass of `rdfs:Class` | `rdfs:Class` and `owl:Class` are equivalent |

# Appendix B

# Databases and Ontologies

Database technology was developed in the early 1970s and has now become the major backbone technology for information systems. Databases gain significant importance as dominant means to store information. In particular, relational databases became the de-facto standard for structured information storage technologies.

## B.1 Relational Databases

Relational databases are today widely used due to their simplicity and the mathematical formal theories that support them. Relational database management systems (RDBMS) are nowadays well understood and are optimized to handle enormous volumes of data. RDBMS has become the dominant data-processing software in use today. The popularity of relational database management systems is based on the simplicity of its concepts, which are easy to communicate between developers and managers, and the emergence of standards like SQL.

Relational databases are based on the relational data model proposed by E.F. Codd in his paper "*A Relational Model of Data for Large Data Banks*" in 1970 [46]. In the relational model, all data are logically structured within relations (tables), that is a set of tuples of atomic attribute values. Attributes are of primitive types, for example: integer, string, real, etc.

The relational model typically allows for the definition of additional consistency rules, such as:

- primary keys, i.e. attributes that uniquely identify a tuple in a relation;
- referential integrity constraints, i.e. the foreign key attributes of a dependent relation are constrained by the values of the primary key of another relation.

The structure of a database is described by its meta data. Metadata contains information about table names, attribute names and types, keys, indices, constraints, etc. This metadata is quite often understood as the formal documentation of a database structure.

In order to describe the concepts of the relational model on a business level, graphical entity-relationship (E/R) modeling technique was introduced by Chen in the 1970s [43]. In E/R diagrams concepts are modeled as entities with attributes. Entities can have relationships with each other, additionally qualified by cardinalities.

In 1978 a study group on database management systems proposed the well-known ANSI/SPARC three-schema architecture that describes the different views of a database [156]. ANSI/SPARC architecture recognizes three views of data: the external view, the internal view, and the conceptual view.

- The *internal view* deals with the physical definition and organization of data.
- The *external view* is concerned with how users view the database. An individual user's view represents the portion of the database that will be accessed by that user.
- The *conceptual view* is an abstract definition of the database. It is "the real world" view of the enterprise being modeled in the database. It represents data and relationships between data without considering the requirements of individual applications or the restrictions of the physical storage media.

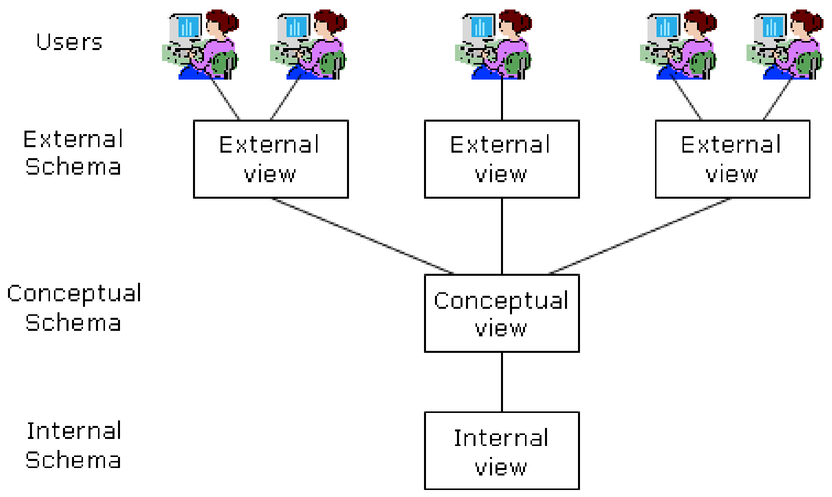


Figure B.1: The ANSI/SPARC Architecture

Figure B.1 demonstrates ANSI/SPARC architecture.

A conceptual schema of a database is quite similar to an ontology. In the next subsection, we review some differences between databases and ontologies.

## B.2 Comparison

In this section we give a comparison between ontologies and databases. This comparison is adopted and combined from [2], [126] and [103].

### 1. Intended use

Intuitively, databases are intended to store information. A conceptual schema of a database provides an abstraction from the physical details. The conceptual model is mainly intended to describe structural information (how objects relate to each other), and integrity constraints (e.g. primary and foreign keys). Data models are at most shared by a limited set of applications.

However, an ontology provides the specification of a domain theory and not the structure of a data container [103]. Ontologies are independent of physical aspects of an implementation. Ontologies represent shared knowledge between several parties.

### 2. Expressiveness

Data modeling and query languages such as SQL allow for the specification of relational models. This is sufficient as long as the intended model is adequately expressible as a relational model.

Ontology data models are syntactically and semantically richer than common approaches for databases. Ontologies typically use a rich logical language to model concepts comprising mechanisms such as synonyms, inheritance, etc. Furthermore, domain rules can be expressed to constrain the set of possible models. The number of representation primitives in many ontologies is much larger than in a typical database schema. For example, many ontology languages and systems allow the specification of cardinality constraints, inverse properties, transitive properties, disjoint classes, and so on. Some languages (e.g., OWL) add primitives to define new classes as unions or intersections of other classes, as an enumeration of its members, as a set of objects satisfying a particular restriction.

**3. Machine processability**

Data models, or their implementations, are highly optimized for executability. Databases nowadays can process large volumes of data with high reliability. Ontologies can be used as simple specification mechanisms and, thus, do not need to be machine-processable (beyond support for editing and presentation). However there can be tool support that allows execution in certain aspects: heavy-weighted ontologies can be supported by inference engines that allow the deriving of new facts from given facts.

**4. Ontologies are data, too**

Ontologies themselves are data to the extent to which database schemas have never been. Ontologies themselves (and not the instance data) are used as controlled vocabularies, to drive search, to provide navigation through large collections of documents, to provide organization and configuration structure of Web sites. A result of a database query is usually a collection of instance data or references to text documents, whereas a result of an ontology query can include elements of the ontology itself (e.g., all subclasses of a particular class).

**5. Ontologies themselves incorporate semantics**

Database schemas often do not provide explicit semantics for their data. Either the semantics has never been specified, or the semantics were specified explicitly at database-design time, but the specification has not become part of database specification and is not available anymore.

Ontologies, however, are logical systems that themselves incorporate semantics. Formal semantics of knowledge-representation systems allow us to interpret ontology definitions as a set of logical axioms.

**6. Ontologies are more often reused**

A database schema defines the structure of a specific database and other databases and schemas do not usually directly reuse or extend existing schemas. The schema is part of an integrated system and is rarely used apart from it. The situation with ontologies is exactly the opposite: Ontologies often reuse and extend other ontologies, and they are not bound to a specific system. An ontology must be a shared and consensual terminology because it is used for information sharing and exchange.

### 7. Ontologies are de-centralized by nature

Traditionally, database schema development and update is a centralized process: Developers of the original schema usually make the changes and maintain the schema. At least, database-schema developers usually know which databases use their schema.

By nature, ontology development is a much more de-centralized and collaborative process. As a result, there is no centralized control over who uses a particular ontology. It is much more difficult to enforce or synchronize updates: If we do not know who the users of our ontology are, we cannot inform them about the updates and cannot assume that they will find out themselves. Lack of centralized and synchronized control also makes it difficult (and often impossible) to trace the sequence of operations that update an ontology.

### 8. Classes and instances can be the same

Databases make a clear distinction between the schema and the instance data. In many rich knowledge-representation systems it is hard to distinguish where an ontology ends and instances begin. The use of metaclasses (classes which have other classes as their instances) in many systems (e.g., Protégé [125], RDFS [35]) blurs or erases completely the distinction between classes and instances. In set-theoretic terms, metaclasses are sets whose elements are themselves sets. This means that "being an instance" and "being a class" is actually just a role for a concept.

In summary, databases and ontologies do not view the world in the same way. However, these techniques need to be related in some contexts (Section 5.1.1). A comparitive overview of both techniques is necessary to understand how we can relate them. Table B.1 summarizes the comparison between databases and ontologies.

Table B.1: Comparison between databases and ontologies

| | **Databases / conceptual schemas** | **Ontologies** |
|---|---|---|
| Intended use | store information, describe structural information and integrity constraints | provide a specification of a domain theory |
| Sharing | shared by at most a limited set of applications | shared between several parties |
| Expressiveness | restricted to the relational model | syntactically and semantically rich, large number of representation primitives |
| Machine-processability | high | limited, (editing tools and inference engines) |
| Query results | a collection of instance data | can include elements of the ontology itself |
| Semantics | do not provide explicit semantics | incorporate semantics themselves |
| Reuse | do not usually directly reuse or extend existing schemas | often reuse and extend other ontologies |
| Development | centralized, synchronized control | de-centralized, no synchronized control |
| Classes and instances | clearly distinct | can be the same |

# Appendix C

# Complete Example on “Associations with SQL statements”

We will use the following example in order to demonstrate how to exploit "*Associations with SQL Statements*" for specifying database-to-ontology mappings. Our example consists of the database and ontology shown in Figure C.1. The database has two principal tables `Paper` and `Author`, and a third table `PaperAuthors` which is used to relate those tables in a many-to-many relationship. The ontology contains two concepts `ex:Paper` and `ex:Author`, which are related to each other using two inverse object properties `ex:writes` and `ex:writtenBy`. Figure C.1 also shows the correspondences between the components of the database and ontology.

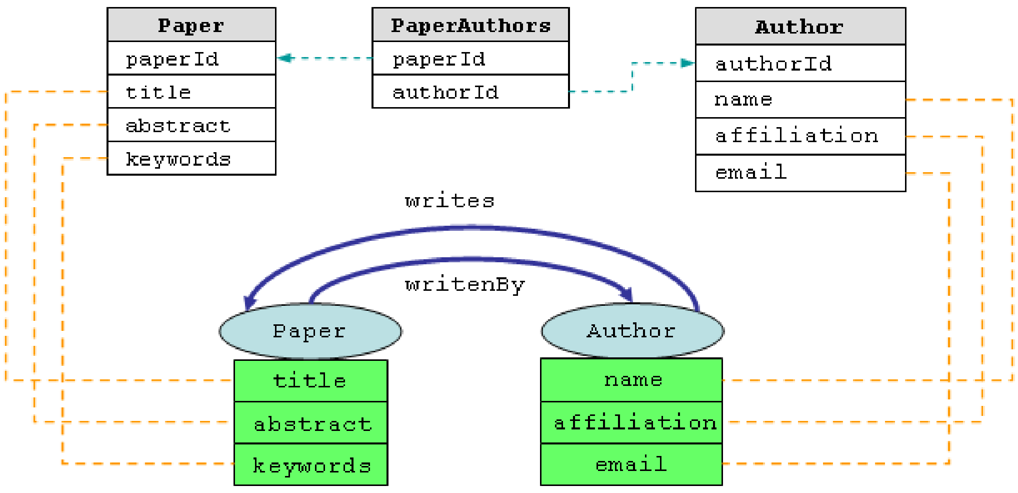


Figure C.1: Complete example: correspondences between a database and on ontology

Using "*Associations with SQL Statements*" we can express these mappings as shown in Table C.1, where each ontology term is associated with the suitable SQL statement.

The complete specification of these mappings in XML format is shown below. This XML format also includes information about the database and ontology being mapped:

Table C.1: Complete example: associations with SQL statements

| **Ontology Term** | **SQL statement** |
|---|---|
| `ex:Paper` | `SELECT Paper.paperId AS DOM FROM Paper` |
| `ex:Author` | `SELECT Author.authorId AS DOM FROM Author` |
| `ex:title` | `SELECT Paper.paperId AS DOM, Paper.title AS RNG` `FROM Paper` |
| `ex:abstract` | `SELECT Paper.paperId AS DOM, Paper.abstract AS RNG` `FROM Paper` |
| `ex:keywords` | `SELECT Paper.paperId AS DOM, Paper.keywords AS RNG` `FROM Paper` |
| `ex:name` | `SELECT Author.authorId AS DOM, Author.name AS RNG` `FROM Author` |
| `ex:affiliation` | `SELECT Author.authorId AS DOM, Author.affiliation AS RNG` `FROM Author` |
| `ex:email` | `SELECT Author.authorId AS DOM, Author.email AS RNG` `FROM Author` |
| `ex:writes` | `SELECT Author.authorId AS DOM, Paper.paperId AS RNG` `FROM Author, Paper, PaperAuthors` `WHERE PaperAuthors.paperId = Paper.paperId` `AND PaperAuthors.authorId = Author.authorId` |
| `ex:writenBy` | `SELECT Paper.paperId AS DOM, Author.authorId AS RNG` `FROM Paper, Author, PaperAuthors` `WHERE PaperAuthors.paperId = Paper.paperId` `AND PaperAuthors.authorId = Author.authorId` |

```
<mappingProject>

  <connectionInfo>
    <driver>com.mysql.jdbc.Driver</driver>
    <url>jdbc:mysql://localhost/dblp</url>
    <user>root</user>
  </connectionInfo>
  <owlOntology>
    <filePath>C:/dblp.owl</filePath>
    <owlURI>http://www.something.com/onto#</owlURI>
  </owlOntology>

  <conceptAssociation>
    <uri>http://www.something.com/onto#Paper</uri>
    <sql>SELECT Paper.paperId AS DOM FROM Paper</sql>
    <dom>Paper.paperId</dom>
  </conceptAssociation>

```

```
<conceptAssociation>
  <uri>http://www.something.com/onto#Author</uri>
  <sql>SELECT Author.authorId AS DOM FROM Author</sql>
  <dom>Author.authorId</dom>
</conceptAssociation>

<datatypePropertyAssociation>
  <uri>http://www.something.com/onto#title</uri>
  <sql>SELECT Paper.paperId AS DOM, Paper.title AS RNG
        FROM Paper</sql>
  <dom>Paper.paperId</dom>
  <rng>Paper.title</rng>
</datatypePropertyAssociation>

<datatypePropertyAssociation>
  <uri>http://www.something.com/onto#abstract</uri>
  <sql>SELECT Paper.paperId AS DOM, Paper.abstract AS RNG
       FROM Paper</sql>
  <dom>Paper.paperId</dom>
  <rng>Paper.abstract</rng>
</datatypePropertyAssociation>

<datatypePropertyAssociation>
  <uri>http://www.something.com/onto#keywords</uri>
  <sql>SELECT Paper.paperId AS DOM, Paper.keywords AS RNG
       FROM Paper</sql>
  <dom>Paper.paperId</dom>
  <rng>Paper.keywords</rng>
</datatypePropertyAssociation>

<datatypePropertyAssociation>
  <uri>http://www.something.com/onto#name</uri>
  <sql>SELECT Author.authorId AS DOM, Author.name AS RNG
       FROM Author</sql>
  <dom>Author.authorId</dom>
  <rng>Author.name</rng>
</datatypePropertyAssociation>

<datatypePropertyAssociation>
  <uri>http://www.something.com/onto#email</uri>
  <sql>SELECT Author.authorId AS DOM, Author.email AS RNG
       FROM Author</sql>
  <dom>Author.authorId</dom>
  <rng>Author.email</rng>
</datatypePropertyAssociation>

<datatypePropertyAssociation>
  <uri>http://www.something.com/onto#affiliation</uri>
  <sql>SELECT Author.authorId AS DOM, Author.affiliation AS RNG
       FROM Author</sql>
```

```
      <dom>Author.authorId</dom>
      <rng>Author.affiliation</rng>
    </datatypePropertyAssociation>

    <objectPropertyAssociation>
      <uri>http://www.something.com/onto#writes</uri>
      <sql>SELECT Author.authorId AS DOM, Paper.paperId AS RNG
           FROM Author, Paper, PaperAuthors
           WHERE PaperAuthors.paperId = Paper.paperId
           AND PaperAuthors.authorId = Author.authorId</sql>
      <dom>Author.authorId</dom>
      <rng>Paper.paperId</rng>
    </objectPropertyAssociation>

    <objectPropertyAssociation>
      <uri>http://www.something.com/onto#writenBy</uri>
      <sql>SELECT Paper.paperId AS DOM, Author.authorId AS RNG
           FROM Paper, Author, PaperAuthors
           WHERE PaperAuthors.paperId = Paper.paperId
           AND PaperAuthors.authorId = Author.authorId</sql>
      <dom>Paper.paperId</dom>
      <rng>Author.authorId</rng>
    </objectPropertyAssociation>
  </mappingProject>
```

# Appendix D

# DOML Mapping Document for Running Example of Chapter 6

```
@prefix doml:    <http://www.doml.org/mapping-specification#> .
@prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#> .
@prefix ont:     <http://www.something.com/ontology#> .
@prefix xsd:     <http://www.w3.org/2001/XMLSchema#> .
@prefix map:     <http://www.something.com/2009/mymap#> .
@prefix rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

# database schema description
map:schema                a     doml:Database ;
      doml:hasTable             map:department-table ,
                                map:person-table ;
      doml:name                 "university" .

map:person-table          a     doml:Table ;
      doml:hasColumn            map:person-firstname-col ,
                                map:person-lastname-col ,
                                map:person-salary-col ,
                                map:person-status-col ,
                                map:person-email-col ,
                                map:person-year-col ,
                                map:person-deptid-col ;
      doml:name                 "person" .

map:department-table        a   doml:Table ;
      doml:hasColumn            map:dept-deptid-col ,
                                map:dept-deptname-col ;
      doml:name                 "department" .

map:person-firstname-col    a    doml:Column ;
      doml:belongsToTable        map:person-table ;
      doml:columnType            xsd:string ;
      doml:isPrimary             "true" ;
      doml:name                  "first_name" .

map:person-lastname-col     a    doml:Column ;
      doml:belongsToTable        map:person-table ;
      doml:columnType            xsd:string ;
      doml:isPrimary             "true" ;
      doml:name                  "last_name" .

map:person-status-col    a       doml:Column ;
      doml:belongsToTable        map:person-table ;
      doml:columnType            xsd:string ;
      doml:name                  "status" .

map:person-email-col     a       doml:Column ;
      doml:belongsToTable        map:person-table ;
      doml:columnType            xsd:string ;
      doml:name                  "email" .
```

```

map:person-year-col      a       doml:Column ;
      doml:belongsToTable        map:person-table ;
      doml:columnType            xsd:string ;
      doml:name                  "year" .

map:person-salary-col   a       doml:Column ;
      doml:belongsToTable       map:person-table ;
      doml:columnType           xsd:decimal ;
      doml:name                 "salary" .

map:person-deptid-col    a       doml:Column ;
      doml:belongsToTable        map:person-table ;
      doml:columnType            xsd:integer ;
      doml:name                  "dept_id" ;
      doml:references            map:dept-deptid-col .

map:dept-deptid-col      a       doml:Column ;
      doml:belongsToTable        map:department-table ;
      doml:columnType            xsd:integer ;
      doml:isPrimary             "true" ;
      doml:name                  "dept_id" .

map:dept-deptname-col    a       doml:Column ;
      doml:belongsToTable        map:department-table ;
      doml:columnType            xsd:string ;
      doml:name                  "dept_name" .

# concept bridges description

map:person-cb            a       doml:ConceptBridge ;
      doml:class                 ont:Person ;
      doml:toTable               map:person-table .

map:student-cb           a       doml:ConceptBridge ;
      doml:class                 ont:Student ;
      doml:toTable               map:person-table ;
      doml:when                  map:status-stud-cond .

map:employee-cb          a       doml:ConceptBridge ;
      doml:class                 ont:Employee ;
      doml:toTable               map:person-table ;
      doml:when                  map:status-emp-cond .

map:department-cb        a       doml:ConceptBridge ;
      doml:class                 ont:Department ;
      doml:toTable               map:department-table .

# datattype property bridges description

```

```
map:student-email-dpb    a          doml:DatatypePropertyBridge ;
      doml:belongsToConceptBridge   map:student-cb ;
      doml:datatypeProperty         ont:email ;
      doml:toColumn                 map:person-email-col ;
      doml:when                     map:status-stud-cond .

map:salary-dpb              a       doml:DatatypePropertyBridge ;
      doml:belongsToConceptBridge   map:employee-cb ;
      doml:datatypeProperty         ont:salary ;
      doml:toColumn                 map:person-salary-col ;
      doml:when                     map:status-emp-cond .

map:employee-name-dpb       a       doml:DatatypePropertyBridge ;
      doml:belongsToConceptBridge   map:employee-cb ;
      doml:datatypeProperty         ont:name ;
      doml:toTransform              map:fn-ln-concat ;
      doml:when                     map:status-emp-cond .

map:person-email-dpb        a       doml:DatatypePropertyBridge ;
      doml:belongsToConceptBridge   map:person-cb ;
      doml:datatypeProperty         ont:email ;
      doml:toColumn                 map:person-email-col .

map:deptname-dpb            a       doml:DatatypePropertyBridge ;
      doml:belongsToConceptBridge   map:department-cb ;
      doml:datatypeProperty         ont:dept-name ;
      doml:toColumn                 map:dept-deptname-col .

map:year-dpb                a       doml:DatatypePropertyBridge ;
      doml:belongsToConceptBridge   map:student-cb ;
      doml:datatypeProperty         ont:year ;
      doml:toColumn                 map:person-year-col ;
      doml:when                     map:status-stud-cond .

map:person-name-dpb         a       doml:DatatypePropertyBridge ;
      doml:belongsToConceptBridge   map:person-cb ;
      doml:datatypeProperty         ont:name ;
      doml:toTransform              map:fn-ln-concat .

map:employee-email-dpb      a       doml:DatatypePropertyBridge ;
      doml:belongsToConceptBridge   map:employee-cb ;
      doml:datatypeProperty         ont:email ;
      doml:toColumn                 map:person-email-col ;
      doml:when                     map:status-emp-cond .

map:student-name-dpb        a       doml:DatatypePropertyBridge ;
      doml:belongsToConceptBridge   map:student-cb ;
      doml:datatypeProperty         ont:name ;
      doml:toTransform              map:fn-ln-concat ;
      doml:when                     map:status-stud-cond .
```

```

# object property bridges description

map:studies-in-opb          a       doml:ObjectPropertyBridge ;
      doml:belongsToConceptBridge   map:student-cb ;
      doml:join-via                 map:person-dept-join ;
      doml:objectProperty           ont:studies-in ;
      doml:refersToConceptBridge    map:department-cb ;
      doml:when                     map:status-stud-cond .

map:works-for-opb           a       doml:ObjectPropertyBridge ;
      doml:belongsToConceptBridge   map:employee-cb ;
      doml:join-via                 map:person-dept-join ;
      doml:objectProperty           ont:works-for ;
      doml:refersToConceptBridge    map:department-cb ;
      doml:when                     map:status-emp-cond .

# transformations and conditions

map:fn-ln-concat    a    doml:Transformation ;
      doml:transType     doml:Concat ;
      doml:hasArguments  map:arglist-fnln.

map:arglist-fnln    a    doml:ArgList;
               rdf:_1    map:person-firstname-col;
               rdf:_2    map:person-lastname-col .


map:arg-list12   a          doml:ArgList ;
                 rdf:_1     map:person-status-col;
                 rdf:_2     "Student".

map:arg-list13   a          doml:ArgList ;
                 rdf:_1     map:person-status-col;
                 rdf:_2     "Employee".

map:status-stud-cond    a   doml:Condition ;
      doml:conditionType    doml:Equals-str ;
      doml:hasArguments     map:arg-list12.

map:status-emp-cond     a   doml:Condition ;
      doml:conditionType    doml:Equals-str ;
      doml:hasArguments     map:arg-list13.

map:arg-list-j   a          doml:ArgList ;
                 rdf:_1     map:person-deptid-col;
                 rdf:_2     map:dept-deptid-col.

map:person-dept-join  a     doml:Condition ;
      doml:conditionType    doml:Equals ;
```

```
        doml:hasArguments     map:arg-list-j.
```

Appendix E

# OWL Ontology for Running Example of Chapter 7

```
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns="http://www.something.com/onto#">
    <owl:Ontology rdf:about="http://www.something.com/onto#"/>

    <owl:Class rdf:about="person"/>
    <owl:DatatypeProperty rdf:about="person.firstName">
        <rdfs:domain rdf:resource="person"/>
        <rdfs:range rdf:resource="&xsd;string"/>
    </owl:DatatypeProperty>
    <owl:DatatypeProperty rdf:about="person.lastName">
        <rdfs:domain rdf:resource="person"/>
        <rdfs:range rdf:resource="&xsd;string"/>
    </owl:DatatypeProperty>

    <owl:Class rdf:about="lecturer">
        <rdfs:subClassOf rdf:resource="person"/>
    </owl:Class>
    <owl:DatatypeProperty rdf:about="lecturer.lecturerRoom">
        <rdfs:domain rdf:resource="lecturer"/>
        <rdfs:range rdf:resource="&xsd;string"/>
    </owl:DatatypeProperty>
    <owl:ObjectProperty rdf:about="lecturer.session">
        <rdfs:domain rdf:resource="lecturer"/>
        <owl:inverseOf rdf:resource="session.lecturer"/>
        <rdfs:range rdf:resource="session"/>
    </owl:ObjectProperty>

    <owl:Class rdf:about="student">
        <rdfs:subClassOf rdf:resource="person"/>
    </owl:Class>
    <owl:FunctionalProperty rdf:about="student.diploma">
        <rdfs:domain rdf:resource="student"/>
        <owl:inverseOf rdf:resource="diploma.student"/>
        <rdfs:range rdf:resource="diploma"/>
    </owl:FunctionalProperty>
    <owl:DatatypeProperty rdf:about="student.studentNumber">
        <rdfs:domain rdf:resource="student"/>
        <rdfs:range rdf:resource="&xsd;string"/>
    </owl:DatatypeProperty>
    <owl:ObjectProperty rdf:about="presence.session">
        <rdfs:domain rdf:resource="student"/>
        <owl:inverseOf rdf:resource="presence.student"/>
        <rdfs:range rdf:resource="session"/>
    </owl:ObjectProperty>

    <owl:Class rdf:about="session"/>
    <owl:FunctionalProperty rdf:about="session.hall">
        <rdfs:domain rdf:resource="session"/>
        <owl:inverseOf rdf:resource="hall.session"/>
        <rdfs:range rdf:resource="hall"/>
    </owl:FunctionalProperty>
    <owl:FunctionalProperty rdf:about="session.lecturer">
        <rdfs:domain rdf:resource="session"/>
        <owl:inverseOf rdf:resource="lecturer.session"/>
        <rdfs:range rdf:resource="lecturer"/>
    </owl:FunctionalProperty>
    <owl:FunctionalProperty rdf:about="session.module">
        <rdfs:domain rdf:resource="session"/>
        <owl:inverseOf rdf:resource="module.session"/>
        <rdfs:range rdf:resource="module"/>
    </owl:FunctionalProperty>
    <owl:DatatypeProperty rdf:about="session.time">
        <rdfs:domain rdf:resource="session"/>
        <rdfs:range rdf:resource="&xsd;date"/>
    </owl:DatatypeProperty>
    <owl:ObjectProperty rdf:about="presence.student">
        <rdfs:domain rdf:resource="session"/>
        <owl:inverseOf rdf:resource="presence.session"/>
        <rdfs:range rdf:resource="student"/>
    </owl:ObjectProperty>

    <owl:Class rdf:about="diploma"/>
    <owl:DatatypeProperty rdf:about="diploma.diploma">
        <rdfs:domain rdf:resource="diploma"/>
        <rdfs:range rdf:resource="&xsd;integer"/>
    </owl:DatatypeProperty>
    <owl:DatatypeProperty rdf:about="diploma.diplomaName">
        <rdfs:domain rdf:resource="diploma"/>
        <rdfs:range rdf:resource="&xsd;string"/>
    </owl:DatatypeProperty>
    <owl:ObjectProperty rdf:about="diploma.module">
        <rdfs:domain rdf:resource="diploma"/>
        <owl:inverseOf rdf:resource="module.diplomaId"/>
        <rdfs:range rdf:resource="module"/>
    </owl:ObjectProperty>
    <owl:ObjectProperty rdf:about="diploma.student">
        <rdfs:domain rdf:resource="diploma"/>
        <owl:inverseOf rdf:resource="student.diploma"/>
        <rdfs:range rdf:resource="student"/>
    </owl:ObjectProperty>
    <owl:Class rdf:about="module"/>
    <owl:FunctionalProperty rdf:about="module.diploma">
        <rdfs:domain rdf:resource="module"/>
        <owl:inverseOf rdf:resource="diploma.module"/>
        <rdfs:range rdf:resource="diploma"/>
    </owl:FunctionalProperty>
    <owl:DatatypeProperty rdf:about="module.module">
        <rdfs:domain rdf:resource="module"/>
        <rdfs:range rdf:resource="&xsd;integer"/>
    </owl:DatatypeProperty>
    <owl:DatatypeProperty rdf:about="module.moduleName">
        <rdfs:domain rdf:resource="module"/>
        <rdfs:range rdf:resource="&xsd;string"/>
    </owl:DatatypeProperty>
    <owl:ObjectProperty rdf:about="module.session">
        <rdfs:domain rdf:resource="module"/>
        <owl:inverseOf rdf:resource="session.module"/>
        <rdfs:range rdf:resource="session"/>
    </owl:ObjectProperty>
    <owl:Class rdf:about="hall"/>
    <owl:DatatypeProperty rdf:about="hall.building">
        <rdfs:domain rdf:resource="hall"/>
        <rdfs:range rdf:resource="&xsd;string"/>
    </owl:DatatypeProperty>
    <owl:DatatypeProperty rdf:about="hall.hallName">
        <rdfs:domain rdf:resource="hall"/>
        <rdfs:range rdf:resource="&xsd;string"/>
    </owl:DatatypeProperty>
    <owl:ObjectProperty rdf:about="hall.session">
        <rdfs:domain rdf:resource="hall"/>
        <owl:inverseOf rdf:resource="session.hall"/>
        <rdfs:range rdf:resource="session"/>
    </owl:ObjectProperty>
</rdf:RDF>
```

# Appendix F

# SPARQL Query Language

We saw in Section A.4 that there are in the literature many ontology specification languages. In particular, markup ontology languages, such as RDF/S and OWL, have currently become the *de facto* of web ontology languages. Along with the development of those languages, works on ontology query languages has been progressing in parallel.

Recently, many languages have been proposed to query web ontologies, such as RQL [96], RDQL [141], and SPARQL [138] for querying RDF/S ontologies, DQL [65] for querying DAML ontologies, and OWL-QL [64] for querying OWL ontologies. In fact, an exhaustive and detailed state of the art on ontology query languages is out of the scope of this dissertation.

However, among these languages, we choose SPARQL to be the ontology query language within OWSCIS system. We use SPARQL language to express ontology queries over both global and local ontologies. Although, SPARQL is mainly intended to query RDF/S models, it can be used to OWL ontologies as well, since OWL builds upon RDF/S. Some of reasons why we choose SPARQL are:

1. It is a W3C standard for an RDF query language (a W3C Recommendation since January 2008).
2. It has a familiar looking SQL-style syntax.
3. Several implementations are available, such as: ARQ, Virtuoso, Sesame, etc.

A SPARQL query contains a set of triple patterns called a *basic graph pattern*. A *triple pattern* is similar to an RDF triple (subject, predicate, object), but any component can be a variable.

We say that a basic graph pattern *matches* a subgraph of the RDF data, when RDF terms from that subgraph may be substituted for the variables. That is, the result of the matching is an RDF graph equivalent to the subgraph.

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

1. **SELECT** – Returns all, or a subset of, the variables bound in a query pattern match.
2. **CONSTRUCT** – Returns an RDF graph constructed by substituting variables in a set of triple templates.
3. **ASK** – Returns a boolean indicating whether a query pattern matches or not.
4. **DESCRIBE** – Returns an RDF graph that describes the resources found.

We are mainly intersted in the SELECT form of SPARQL queries. A SPARQL SELECT query consists of two parts:

1. the SELECT clause that identifies the variables to appear in the query results, and
2. the WHERE clause that provides the basic graph pattern to match against the data graph.

For example, let us consider the following simple SPARQL query that returns the given name of person whose family name is "Smith".

```
PREFIX vCard: <http://www.w3.org/2001/vcard-rdf/3.0#>
SELECT ?givenName
WHERE {
        ?person vCard:Family "Smith" .
        ?person vCard:Given  ?givenName .
}
```

The first line of the query simply defines a PREFIX for the `vCard` namespace, so that we don't have to type it in full each time it is referenced. The SELECT clause specifies what the query should return (in this case, a variable named givenName). SPARQL variables are prefixed with `"?"`.

The WHERE clause consists of a series of triple patterns, expressed using Turtle-based syntax [1]. These triples together comprise what is known as a graph pattern.

The query attempts to match the triples of the graph pattern against the RDF data model. Matching means find a set of bindings such that the substitution of variables for values creates a triple that is in the set of triples making up the RDF graph. Each matching binding of the graph pattern variables to the model nodes becomes a query solution, and the values of the variables named in the SELECT clause become part of the query results.

SPARQL language allows to pose restrictions on the values in query solutions. These restrictions are defined in FILTER clauses. These clauses are, to some extent, similar to WHERE clause of an SQL query.

SPARQL filters can be used to restrict the values of strings (using `REGEX` function), numeric values (using <, >, =, <=, >= and != operators). SPARQL also provides test functions, including `BOUND`, `isURI`, `isBLANK`, and `isLITERAL`, for other term constraints.

For example, in the following SPARQL query, the FILTER clause restrictes the value of the variable `?age` to be less or equal 24.

```
PREFIX info : <http://somewhere/peopleInfo#>
SELECT ?resource
WHERE {
     ?resource    info:age    ?age .
     FILTER (?age <= 24)
}
```

SPARQL also allows to define OPTIONAL blocks that offer the ability to query for data but not to fail query when that data does not exist. That is, optional blocks define additional graph patterns that do bind to the graph when they can be matched, but do not cause solutions to be rejected if they are not matched.

For example, the following query gets the name of a person and also his age if that piece of information is available.

```
PREFIX info:  <http://somewhere/peopleInfo#>
PREFIX vcard: <http://www.w3.org/2001/vcard-rdf/3.0#>
SELECT ?name ?age
WHERE {
    ?person vcard:FN  ?name .
    OPTIONAL { ?person info:age ?age }
}
```

[1] http://www.w3.org/TeamSubmission/turtle/

## SPARQL Query Results XML Format

SPARQL allows query results to be returned as XML, in a simple format known as the SPARQL Query Results XML Format [12]. The SPARQL Results Document begins with sparql root element written as follows:

```
<?xml version="1.0"?>
<sparql xmlns="http://www.w3.org/2005/sparql-results#">
 ...
</sparql>
```

Inside the `sparql` element are two sub-elements, `head` and `results` which must appear in that order.

For a variable binding query result, `head` must contain a sequence of elements describing the set of Query Variable names in the Solution Sequence (query results). The order of the variable names in the sequence is the order of the variable names given to the argument of the `SELECT` statement in the SPARQL query

Inside the `head` element, the ordered sequence of variable names chosen are used to create empty child elements `variable` with the variable name as the value of an attribute `name`:

```
<sparql xmlns="http://www.w3.org/2005/sparql-results#">
  <head>
    <variable name="x"/>
    <variable name="name"/>
  </head>
  ...
</sparql>
```

The second child-element of `sparql` is `results`, and must appear after `head`. The `results` element contains the complete sequence of query results. For each Query Solution in the query results, a `result` child-element of `results` is added:

```
<sparql xmlns="http://www.w3.org/2005/sparql-results#">
  ...  head ...
  <results>
    <result> ...
    </result>
    <result> ...
    </result>
    ...
  </results>
</sparql>
```

Each `result` element corresponds to one Query Solution in a result and contains child elements (in no particular order) for each Query Variable that appears in the solution. It is used to record how the query variables bind to RDF Terms.

Each binding inside a solution is written as an element binding as a child of result with the query variable name as the value of the name attribute. So for a result binding three variables `name`, `hpage` and `age` it would look like:

```
<sparql xmlns="http://www.w3.org/2005/sparql-results#">
  <head>
    <variable name="name"/>
    <variable name="hpage"/>
    <variable name="age"/>
  </head>
  <results>
    <result>
      <binding name="name"> ... </binding>
      <binding name="hpage"> ... </binding>
      <binding name="age"> ... </binding>
    </result>
    <result>
      <binding name="name"> ... </binding>
      <binding name="hpage"> ... </binding>
      <binding name="age"> ... </binding>
    </result>
    ...
  </results>
</sparql>
```

The value of a query variable binding, which is an RDF Term, is included as the content of the binding as follows:

- RDF URI Reference U
  `<binding><uri>U</uri></binding>`
- RDF Literal S
  `<binding><literal>S</literal></binding>`
- RDF Typed Literal S with datatype URI D
  `<binding><literal datatype="D">S</literal></binding>`

An example of a query solution encoded in this format is as follows:

```
<?xml version="1.0"?>
<sparql xmlns="http://www.w3.org/2005/sparql-results#">
  <head>
    <variable name="name"/>
    <variable name="hpage"/>
    <variable name="age"/>
  </head>
  <results>
    <result>
      <binding name="name">
        <literal>Bob</literal>
      </binding>
      <binding name="hpage">
        <uri>http://work.example.org/bob/</uri>
      </binding>
      <binding name="age">
        <literal datatype="http://www.w3.org/2001/XMLSchema#integer">
          30
        </literal>
      </binding>
    </result>
    ...
  </results>
</sparql>
```

# Appendix G

# Background on SPARQL-to-SQL Translation

SQL is a sophisticated query language that has been developed throughout the years from a simple query language based on the relational calculus to a powerful language for integrating data from across multiple data sources. SPARQL is an emerging query language for RDF data and is still rather rudimental. Thus, comparing the expressiveness of a premature query language like SPARQL with a sophisticated language like SQL, would rather be unfair [54].

However, some works show that the expressiveness of SPARQL approaches that of basic SQL. In [54], de Laborda and Conrad worked on bringing relational data into the Semantic Web using a combination of Relational.OWL (see Section 5.2.3) as a Semantic Web representation of relational databases and a semantic query language like SPARQL. They analyze whether all the possible queries on a relational database can be expressed (based on the Relational.OWL representation of that specific database) using SPARQL query language. To do that, they simulate the most important expressions of SQL (which can be represented by the basic operations of the relational algebra: selection, projection, set union, set difference and Cartesian product) with SPARQL queries. This analysis shows that the combination of Relational.OWL and SPARQL is relational complete.

In our investigation about existing works on SPARQL-to-SQL translation, we found only some works that handle SPARQL query translation into equivalent SQL query over RDF triple stores. We do not find any work that handle SPARQL query translation into SQL over arbitrary relational database.

An RDF triple store is a database devoted to store RDF triples. In the literature, there are many projects that propose to use relational databases to store RDF data. W3C has made a survey on these projects in [13], such as: Jena, KAON, Sesame, RDF Suite, etc. These projects use different database schemas dedicated for triple stores.

A widely-used scheme for storing RDF statements in a relational database is the standard triple store. In this approach, an RDF graph can be represented as a `"Triples"` table with three columns: `subject`, `predicate` and `object`. Each row corresponds to one triple/statement. For example, the following RDF graph:

```
ex:Alice    rdf:type    foaf:Person
ex:Alice    foaf:name   "Alice"
ex:Alice    ex:age      "33"
ex:Bob      rdf:type    foaf:Person
ex:Bob      foaf:name   "Bob"
ex:Bob      foaf:mbox   <mailto:bob@example.org>
ex:Bob      ex:age      "42"
```

can be represented as the following table:

| **subject** | **predicate** | **object** |
|---|---|---|
| `ex:Alice` | `rdf:type` | `foaf:Person` |
| `ex:Alice` | `ex:name` | `"Alice"` |
| `ex:Alice` | `ex:age` | `"33"` |
| `ex:Bob` | `rdf:type` | `foaf:Person` |
| `ex:Bob` | `ex:name` | `"Bob"` |
| `ex:Bob` | `ex:mbox` | `<mailto:bob@example.org>` |
| `ex:Bob` | `ex:age` | `"42"` |

The existing works on SPARQL-to-SQL query translation (such as [52] and [42]) aim at translating a SPARQL query over a given RDF graph into an SQL query over the triple store database of this RDF graph. In the following subsections, we present two of the most significant works.

## G.1 A Relational Algebra for SPARQL

Cyganiak in [52] defines a relational algebra for SPARQL and outlines a set of rules to establish the equivalence between this algebra and SQL. This algebra consists of several operations (similar to those of usual relational algebra) such as: selection, projection/rename, inner join, left outer join, union, and set difference.

The first step of translation process is to express each SPARQL triple pattern as a relational operation. This operation can be obtained from the graph relation (table `"Triples"`) by constant attribute selection on the concrete (non-variable) nodes and by projection/renaming on the variable nodes.

For example, the triple `{?person rdf:type foaf:Person}` has two concrete nodes: `rdf:type` and `foaf:Person`, and a variable node `?person`. The translation of this triple pattern is a selection over the concrete nodes: `?predicate='rdf:type'` and `?object='foaf:Person'`, and a projection on the attribute `?subject` renamed to: `?person`.

$$R_1 = \Pi_{?person \leftarrow ?subject}\left(\sigma_{?predicate=rdf:type \wedge ?object=foaf:Person}\left(Triples\right)\right)$$

The next step is to join the relational operations translated from single triple patterns into a complex relational operation for the query pattern. The join is done in this order:

1. The relational operations of all triple patterns are inner joined (their relative order does not matter).
2. Then, all OPTIONAL subpatterns are left outer joined to the result of 1, in the order they occur in the query, with the optional subpattern becoming the right side of the outer join.
3. Finally, all FILTER conditions are applied as selection operators to the result of 2 (their relative order does not matter).

The work of Cyganiak does not provide a solution to the case of nested OPTIONAL patterns.

For example, the following SPARQL query:

```
SELECT ?name ?email
WHERE {
    ?person   rdf:type    foaf:Person .
    ?person   foaf:name   ?name .
    OPTIONAL { ?person  foaf:mbox  ?email }
}
```

is translated to the following relational operators tree:

This tree can be simplified into the SQL expression:

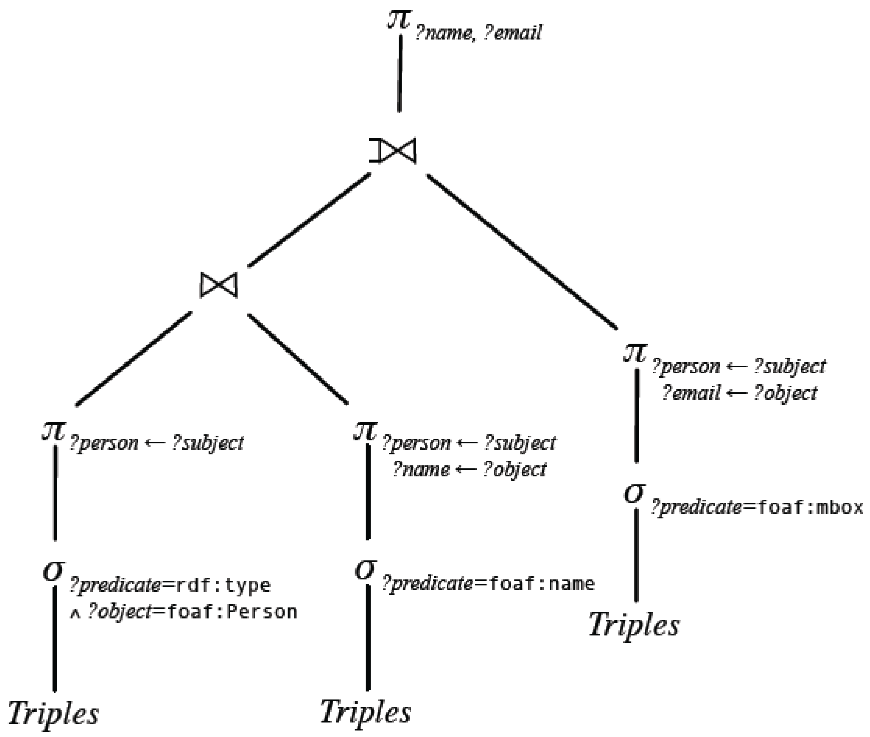

π ?name, ?email
⟕
⋈
π ?person ← ?subject
σ ?predicate=rdf:type ∧ ?object=foaf:Person
Triples
π ?person ← ?subject ?name ← ?object
σ ?predicate=foaf:name
Triples
π ?person ← ?subject ?email ← ?object
σ ?predicate=foaf:mbox
Triples

FIGURE G.1: Relational Tree for Cyganiak's Relational Algebra for SPARQL

```
   SELECT sub1.name, sub3.email
     FROM
  ((SELECT t1.subject AS person
     FROM Triples AS t1
    WHERE t1.predicate='rdf:type'
      AND t1.object='foaf:Person')
       AS sub1
     JOIN
   (SELECT t2.subject AS person,
           t2.object AS name
     FROM Triples AS t2
    WHERE t2.predicate='foaf:name')
       AS sub2
       ON sub1.person=sub2.person)
LEFT JOIN
   (SELECT t3.subject AS person,
           t3.object AS email
     FROM Triples AS t3
    WHERE t2.predicate='foaf:mbox')
       AS sub3
       ON sub1.person=sub3.person
```

This SQL expression is also flattened into this expression:

```
   SELECT t1.object AS name, t3.object AS email
     FROM Triples AS t1
     JOIN (Triples AS t2)
       ON t1.subject=t2.subject
LEFT JOIN (Triples AS t3)
       ON t1.subject=t3.subject
      AND t1.predicate='foaf:name'
      AND t3.predicate='foaf:mbox'
      AND t2.predicate='rdf:type'
      AND t2.object='foaf:Person'
```

## G.2 Semantics Preserving SPARQL-to-SQL Query Translation

Chebotko et al. [42] propose another SPARQL-to-SQL translation that provides a solution to the nested OPTIONAL pattern problem. This work consists of two algorithms: the first one, `BGPtoSQL`, translates a basic graph pattern to its SQL equivalent. The second algorithm, `SPARQLtoSQL`, is based on BGPtoSQL, and propose a semantics preserving SPARQL-to-SQL query translation for SPARQL queries that contain arbitrary complex optional graph patterns.

These algorithms use two data structures, which are: the basic graph pattern model and SPARQL query model.

- *Basic Graph Pattern Model* – A basic graph pattern is modeled as a directed graph $BGP = (N, E)$, where $N$ is a set of nodes representing subjects and objects, and $E$ is a set of edges representing predicates. Each edge is directed from a subject node to an object node. Each node is labeled with a variable name, a URI, or a literal, and each edge is labeled with a variable name or a URI.
- *SPARQL Query Model* – A SPARQL query $Q$ is modeled as a tuple $(S_{sel}, T)$, where $S_{sel}$ is the ordered list of variables in the SELECT clause and $T = (N, E)$ is the ordered tree of clauses rooted at the WHERE clause of $Q$, where $N$ is a set of nodes representing clauses in $Q$, and $E$ is a set of edges representing the containment relationship between two clauses, such that there exists an edge between nodes $n_1$ and $n_2$ if and only if the clause of $n_2$ is directly contained (nested) in the clause of $n_1$.

### BGPtoSQL

The algorithm firstly constructs the FROM and SELECT clauses of the target SQL query. For each edge in the BGP, a unique alias of the table Triples is generated, and all the aliases are added to the FROM clause. All distinct variables in the BGP are projected in the SELECT clause, such that a predicate/subject/object variable is represented by the corresponding column of the Triples table.

Then, the WHERE clause is constructed. First, all restrictions with respect to labeling are included (nodes that are URIs or literals). Second, shared variables require unique instantiation and thus must participate in join conditions. For example, if a node is labeled with a variable and has an incoming edge and an outgoing edge, then the object attribute of the incoming edge table alias must be equal to the subject attribute of the outgoing edge table alias.

## SPARQLtoSQL

First, for each basic graph pattern $BGP_i$ of the query, the `BGPtoSQL` algorithm is used to generate an equivalent SQL query. The result is stored in relation $R_i$.

Second, when joining two relations, left outer join ($:\bowtie$) is used to preserve all tuples of the first relation in the resulting relation[1].

Third, to preserve the left-associativity semantics of OPTIONAL clauses, the strategy employs the preorder traversal of tree $T$ to join the relations, such that each visited edge corresponds to a left outer join. For example, the first join is always between the relation $R_r$ that corresponds to the root of the SPARQL query model and the relation that corresponds to the root' first child. The result of this join is denoted as $R_{res}$. Next, $R_{res}$ is joined with the root' first grandchild if any, otherwise, with the root' second child. Again, the result is assigned to $R_{res}$. After each join, all distinct relational attributes are projected, since they may participate in following join conditions. After the last join, the attributes that correspond to the variables in the SPARQL SELECT clause are projected.

[1]The left outer join captures the basic semantics of OPTIONAL patterns: the evaluation of an OPTIONAL clause is not obligated to succeed, and in the case of failure, a NULL value will be returned for unbound variables.

# Appendix H

# Algorithms for SPARQL-to-SQL Translation

This appendix contains the complete algorithms mentioned in our SPARQL-to-SQL Translation methods (Chapter 8). These algorithmes are:

- For translation using “Associations with SQL Statements”:
  - Query Parsing Algorithm (Section H.1.1)
  - Query Translation Algorithm (Section H.1.2).
  - Query Simplification Algorithm (Section H.1.3).
- For translation using DOML mappings:
  - Preprocessing Algorithm (Section H.2.1).
  - Query Translation Algorithm (Section H.2.2).

## H.1 Algorithms for Translation using “Associations with SQL Statements”

### H.1.1 Query Parsing Algorithm

TABLE H.1: Query Parsing Algorithm

```
1.  Algorithm QueryParse
2.  Input: SPARQL query $Q_{SPARQL}$
3.  Output: set of selected variables $SV$, set of order variables $OV$,
4.           set of filters $FLTR$, and basic graph pattern $BGP = (N, E)$
5.  Begin
6.    If ($Q_{SPARQL}$ is QueryResultStar)
7.      SV = $Q_{SPARQL}$.getAllVariables()
8.    Else
9.      SV = $Q_{SPARQL}$.getSelectedVariables()
10.   End If
11.   $OV = Q_{SPARQL}$.getOrderVaribles()
12.   $Elements = Q_{SPARQL}$.getPatternElements()
13.   For each $elem \in Elements$
14.     If ($elem$ is filter)
15.       FLTR += $elem$.asFilter()
16.     Else If ($elem$ is triple pattern)
17.       triple = $elem$.asTriple()
18.       subjectNode = triple.getSubject()
19.       predicateNode = triple.getPredicate()
20.       objectNode = triple.getObject()
21.       $v_{sub}$ = createVertex()
22.       $v_{obj}$ = createVertex()
23.       $v_{sub}$.setLabel(subjectNode)
24.       $v_{obj}$.setLabel(objectNode)
```

```
25.             If $v_{sub} \notin N$ Then $N+ = v_{sub}$
26.             If $v_{obj} \notin N$ Then $N+ = v_{obj}$
27.             $e_{prd}$ = createEdge(subjectNode, objectNode)
28.             $e_{prd}$.setLabel(predicateNode)
29.             $e_{prd}$.setAlias(new unique alias)
30.             $E+ = e_{prd}$
31.         End If
32.     End For
33. End
```

### H.1.2 Query Translation Algorithm

TABLE H.2: Query Translation Algorithm

```
1.  Algorithm SPARQL2SQL
2.  Input: a set of selected variables $SV$, a set of order variables $OV$,
3.           a set of filters $FLTR$, a basic graph pattern $BGP = (N, E)$, and
4.           a mapping documunt $MD$
5.  Output: $\mathbb{SELECT}$: a set of SELECT items of the translated SQL query,
6.           $\mathbb{FROM}$: a set of FROM items of the translated SQL query, and
7.           $\mathbb{WHERE}$: a set of WHERE items of the translated SQL query.
8.  Begin
9.      $\mathbb{SELECT} = \{\}$
10.     $\mathbb{FROM} = \{\}$
11.     $\mathbb{WHERE} = \{\}$
12.
13.     /* Construct FROM clause */
14.     For each edge $e \in E$
15.         alias = e.getAlias()
16.         prd = e.getLabel()
17.         If (prd = "rdf:type")
18.             targetNode = edge.getTargetNode()
19.             assoc = MD. getAssociation(targetNode)
20.             $\mathbb{FROM}+ =$ "(" + assoc.getSQL + ") AS " + alias
21.         Else
22.             assoc = MP.getAssociation(prd)
23.             $\mathbb{FROM}+ =$ "(" + assoc.getSQL + ") AS " + alias
24.         End If
25.     End For
26.
27.     /* Construct SELECT clause */
28.     For each node $v \in V$
29.         If (v is a varibale)
```

```
30.             var = v.asVariable().getVariableName()
31.         If (v.in-degree> 0)
32.             Let $e_1^{in}$ be the first incoming edge of v
33.             var.setAliasedColumn($e_1^{in}$.getAlias +".RNG")
34.             If $(var \in SV)$
35.                 $\mathbb{SELECT}+=e_1^{in}$.getAlias +".RNG AS "+var
36.             End If
37.         Else If (v.out-degree> 0)
38.             Let $e_1^{out}$ be the first outgoing edge of v
39.             var.setAliasedColumn($e_1^{out}$ .getAlias +".DOM")
40.             If $(var \in SV)$
41.                 $\mathbb{SELECT}+=e_1^{out}$.getAlias +".DOM AS "+var
42.             End If
43.         End If
44.     End If
45. End For
46.
47. /* Construct WHERE clause*/
48. For each node $v \in V$
49.     If ($v$ is a varibale)
50.         If ($v$.in-degree $> 1$)
51.             Let $e_1^{in}$ be the first incoming edge of $v$
52.             $x = e_1^{in}$.getAlias() +".RNG"
53.             For each incoming edge $e^{in}$ of $v$
54.                 $y = e^{in}$.getAlias() +".RNG"
55.                 $\mathbb{WHERE}+= x +$ " = " $+ y$
56.             End For
57.         End If
58.         If ($v$.out-degree $> 1$)
59.             Let $e_1^{out}$ be the first outgoing edge of $v$
60.             $x = e_1^{out}$.getAlias +".DOM"
61.             For each incoming edge $e^{out}$ of $v$
62.                 $y = e^{out}$.getAlias() +".DOM"
63.                 $\mathbb{WHERE}+= x +$ " = " $+ y$
64.             End For
65.         End If
66.         If (v.in-degree $> 0$ && v.out-degree $> 0$)
67.             Let $e_1^{in}$ be the first incoming edge of $v$
68.             Let $e_1^{out}$ be the first outgoing edge of $v$
69.             $x = e_1^{in}$.getAlias() +".RNG"
70.             $y = e_1^{out}$.getAlias() +".DOM"
71.             $\mathbb{WHERE}+= x +$ " = " $+ y$
72.         End If
73.     Else If (v is literal)
```

```
74.             For each incoming edge $e^{in}$ of node
75.                 $\mathbb{WHERE}+ = e^{in}$.getAlias() +".RNG = " + $v$.asLiteral()
76.             End For
77.         End If
78.     End For
79.     For each filter $FL \in FLTR$
80.         expr = filter2sql(FL)
81.         Let $V_{FL}$ be the set of variables mentioned in $FL$
82.             For each variable $var \in V_{FL}$
83.                 $ac = var$.getAliasedColumn()
84.                 $expr$.replace($var$, $ac$)
85.             End For
86.         $\mathbb{WHERE}+ = expr$
87.     End For
88. End
```

### H.1.3 Query Simplification Algorithm

Table H.3: Query Simplification Algorithm

```
1.  Algorithm SQL-simplification
2.  Input: $\mathbb{SELECT}$: a set of SELECT items of the translated SQL query,
3.             $\mathbb{FROM}$: a set of FROM items of the translated SQL query, and
4.             $\mathbb{WHERE}$: a set of WHERE items of the translated SQL query.
5.  Output: a simplified SQL query $SQL_{simple}$
6.  Begin
7.      $SELECT_{final} = \phi$
8.      $FROM_{final} = \phi$
9.      $WHERE_{final} = \phi$
10.     For each outer FROM item $FI_{out} \in \mathbb{FROM}$
11.         SA = $FI_{out}$. getAlias()
12.         subSQL = $FI_{out}$.getSQL() /* the item is itself an SQL statement */
13.         $SELECT_{inner}$ = subSQL.getSELECT()
14.         $FROM_{inner}$ = subSQL.getFROM()
15.         $WHERE_{Einner}$ = subSQL.getWHERE()
16.         For each inner SELECT item $SI_{in} \in SELECT_{inner}$
17.             $ac = SA$ + "." + $SI$.getAlias()            /* aliased column */
18.             $rc = SI_{in}$.getTable()+ "." + $SI_{in}$.getColumn() /* real column */
19.             $CM$.put($ac$, $rc$)
20.         End For
21.         For each inner FROM item $FI_{in} \in FROM_{inner}$
22.             If ($FI_{in} \notin FROM_{final}$) Then $FROM_{final}+ = FI_{in}$
23.         End For
```

```
24.         For each inner WHERE item $WI_{in} \in WHERE_{inner}$
25.             $WHERE_{final} += WI_{in}$
26.         End For
27.     End For
28.     For each outer SELECT item $SI_{out} \in \mathbb{SELECT}$
29.         $alias = SI_{out}.$ getAlias()
30.         $ac = SI_{out}.$ getItem()          /* aliased column */
31.         $rc = CM$.get($ac$)              /* real column */
32.         $SELECT_{final} += rc$ + " AS " + $alias$
33.     End For
34.
35.     For each outer WHERE item $WI_{out} \in \mathbb{WHERE}$
36.         For each aliased column $ac$ mentioned in $WI_{out}$
37.             $rc$ = CM.get($ac$)
38.             $WI_{out}.$ replace($ac$, $rc$)
39.             $WHERE_{final} += WI_{out}$
40.         End For
41.     End For
42.     $WHERE_{temp} = \phi$
43.     For each final WHERE item $WI_{final} \in WHERE_{final}$
44.         If ($WI_{final}$ is not trivial)
45.             If ($WI_{final} \notin WHERE_{temp}$)
46.                 $WHERE_{temp} += WI_{final}$
47.             End If
48.         End If
49.     End For
50.     $WHERE_{final} = WHERE_{temp}$
51.     $SQL_{simple}$ = "SELECT " + $SELECT_{final}$
52.         + " FROM " + $FROM_{final}$
53.         + " WHERE " + $WHERE_{final}$
54. End
```

## H.2 Algorithms for Translation using DOML mappings

### H.2.1 Preprocessing Algorithm

Table H.4: Preprocessing Algorithm

```
1.  Function Preprocessing
2.      $v2pcSet = \phi$
3.      For each edge $e \in E$
4.          $prdNode = e$.getLabel()
5.          $subNode = e$.getSource()
```

```
6.          $objNode = e$.getTarget()
7.          **If**(subNode.isVariable())
8.              $subVar = subNode$.asVariable()
9.              **If** $prdNode$ = "rdf:type"
10.                 **If** $objNode$.isURI()
11.                     $pc = [objNode]$
12.                     $v2pc$ =**new** VAR2PossCon($subVar$, $pc$)
13.                     $v2pcSet+ = v2pc$
14.                 **End If**
15.             **Else If** $prdNode$ is a datatype property
16.                 $pc = \phi$
17.                 $DPBs$ = domlParser.getDPBs($prdNode$)
18.                 **For** each $\beta \in DPBs$
19.                     $cb = \beta$.getDomCB()
20.                     $pc+ = cb$.getConcept()
21.                 **End For**
22.                 $v2pc$ = **new** VAR2PossCon ($subVar$, $pc$)
23.                 $v2pcSet+ = v2pc$
24.             **Else If** $prdNode$ is an object property
25.                 $pc_1 = \phi$
26.                 $pc_2 = \phi$
27.                 $OPBs$ = domlParser.getOPBs($prdNode$)
28.                 **For** each $\delta \in OPBs$
29.                     $pc_1+ = \delta$.getDomCB().getConcept()
30.                     **If** $objNode$ is variable
31.                         $pc_2+ = \delta$.getRngCB().getConcept()
32.                     **End If**
33.                 **End For**
34.                 $v2pc_1$ =**new** VAR2PossCon($subVar$, $pc_1$)
35.                 $v2pcSet+ = v2pc_1$
36.                 **If** $objNode$ is variable
37.                     $objVar = objNode$.asVariable()
38.                     $v2pc_2$ = **new** VAR2PossCon($objVar$, $pc_2$)
39.                     $v2pcSet+ = v2pc_2$
40.                 **End If**
41.             **End If**
42.         **End If**
43.     **End For**
44.
45.     **For** each $v2pc \in v2pcSet$
46.         $var = v2pc$.getVar()
47.         $pc = v2pc$.getPossibleConcepts()
48.         **If** ($v2pcMap$.containsKey($var$))
49.             $oldPC = v2pcMap$.get($var$)
```

```
            $newPC = oldPC \cap pc$
            $v2pcMap$.put($var$, $newPC$)
        Else
            $v2pcMap$.put($var$, $pc$)
        End If
    End For

    For each $v2pc \in v2pcMap$
        $v = v2pc$.getVar()
        $pc = v2pc$.getPossibleConcepts()
        If $pc$.size $= 1$
            $var2concept$.put($v$, $pc[1]$)
        Else
            Let $C$ be the most general concept in $pc$
            $var2concept$.put($v$, $C$)
        End If
    End For
Return $var2concept$
End Function
```

### H.2.2 Query Translation Algorithm

```
Begin
    $\mathbb{SELECT} = \phi$
    $\mathbb{FROM} = \phi$
    $\mathbb{WHERE} = \phi$
    $var2column$ = new map
    For each $e \in E$
        $prdNode = e$.getLabel()
        $subNode = e$.getSource()
        $objNode = e$.getDestination()
        If subNode is variable Then $subVar = subNode$.asVariable()
        If objNode is variable Then $objVar = objNode$.asVariable()
        If $prdNode$ is rdf:type
            $con = objNode$
            $cb = domlParser$.getConceptBridge($con$)
            $table = cb$.getTable()
            addFromItem($table$)
            If cb has a condition
                $condID = cb$.getCondition()
                $condition = domlParser$.getCondByID($condID$)
                addCondition($condition$)
            End If
```

```
22.         Else If $prdNode$ is a datatype property
23.             $DPBs = domlParser$.getDPBs($prdNode$)
24.             For each $\beta \in DPBs$
25.                 $con = \beta$.getDomCB().getConcept()
26.                 If $con = var2concept$.get($subVar$)
27.                     $dpb = \beta$
28.                 End If
29.             End For
30.             If $dpb$ has a column
31.                 $colID = dpb$.getColumn()
32.                 $col = domlParser$.getColumnByID($colID$)
33.                 addFromItem($col$.getTableName())
34.                 If $objNode$ is variable
35.                     $var2column$.put($objVar$, $col$.fullName())
36.                 Else If $objNode$ is literal
37.                     $value = objNode$.getLiteralValue()
38.                     $exp = col$.fullName() + " = " + $value$
39.                     addWhereItem($exp$)
40.                 End If
41.             Else If $dpb$ has a transformation
42.                 $transID = dpb$.getTransform()
43.                 $transform = domlParser$.getTransByID($transID$)
44.                 $TS$ =trans2SQL($transform$)
45.                 If $objNode$ is variable
46.                     $var2column$.put($objVar$, $TS$)
47.                 Else If $objNode$ is literal
48.                     $val = objNode$.getLiteralValue()
49.                     $exp = TS+$" = " $+val$
50.                     addWhereItem($exp$)
51.                 End If
52.                 Let $mc$ be the set of columns mentioned in the transformation
53.                 For each column $col \in mc$
54.                     $tableName = col$.getTableName()
55.                     addFromItem($tableName$)
56.                 End For
57.             End If
58.             If $dpb$ has a condition
59.                 $condID = dpb$.getCondition()
60.                 $condition = domlParser$.getCondByID($condID$)
61.                 addCondition($condition$)
62.             End If
63.         Else If $prdNode$ is an object property
64.             $OPBs = domlParser$.getOPBs($prdNode$)
65.             For each $\delta \in OPBs$
```

```
66.                     $con = \delta$.getDomCB().getConcept()
67.                     If ($con == var2concept$.get($subVar$))
68.                         $opb = \delta$
69.                     End If
70.                 End For
71.                 $joinID = opb$.getJoin()
72.                 $join = domlParser$.getCondByID($joinID$)
73.                 addCondition($join$)
74.                 If $opb$ has a condition
75.                     $condID = opb$.getCondition()
76.                     $condition = domlParser$.getCondByID($condID$)
77.                     addCondition($condition$)
78.                 End If
79.             End If
80.         End For
81.         For each $v \in SV$
82.             $col = var2column$.get($v$)
83.             $\mathbb{SELECT} += col +$ " AS " $+ v$
84.         End For
85.         For each filter $FL \in FLTR$
86.             $expr =$filter2SQL($FL$)
87.             Let $VFL$ be the set of variables mentioned in $FL$
88.             For each variable $var \in VFL$
89.                 $col = var2column$.get($var$)
90.                 $expr$.replace($var$, $col$)
91.             End For
92.             AddWhereItem($expr$)
93.         End For
94.     End
95.     Procedure addFromItem($fi$)
96.         If $fi \notin \mathbb{FROM}$
97.             $\mathbb{FROM} += fi$
98.         End If
99.     End Procedure
100.    Procedure addWhereItem($wi$)
101.        If $wi \notin \mathbb{WHERE}$
102.            $\mathbb{WHERE} += wi$
103.        End If
104.    End Procedure
105.    Procedure addCondition($cond$)
106.        $exp =$cond2Expression($cond$)
107.        addWhereItem($exp$)
108.        Let $mc$ be the set of columns mentioned in $exp$
109.        For each column $col \in mc$
```

```
110.          addFromItem($col$.getTableName())
111.       End For
112.    End Procedure
113.    Function trans2SQL($trans$)
114.       $operator = trans$.getTransType()
115.       $ops = \phi$
116.       For each $arg \in trans$.getArguments()
117.          If $arg$ is Constant
118.             $operand = arg$
119.          Else If $arg$ is Column
120.             $col = arg$.asColumn()
121.             $operand = col$.fullName()
122.          Else If $arg$ is Transformation
123.             $trans_1 = arg$.asTransformation()
124.             $operand =$trans2SQL($trans_1$)
125.          End If
126.          $ops+ = operand$
127.       End for
128.       $exp =$ new Expression($operator$, $ops$)
129.       Return $exp$
130.    End Function
131.    Function condition2SQL($cond$)
132.       $operator = cond$.getConditionType()
133.       $ops = \phi$
134.       For each $arg \in cond$.getArguments()
135.          If $arg$ is Constant
136.             $operand = arg$
137.          Else If $arg$ is Column
138.             $col = arg$.asColumn()
139.             $operand = col$.fullName()
140.          Else If $arg$ is Condition
141.             $cond_1 = arg$.asCondition()
142.             $operand =$condition2SQL($cond_1$)
143.          Else If $arg$ is Transformation
144.             $trans = arg$.asTransformation()
145.             $operand =$trans2SQL($trans$)
146.          End If
147.          $ops+ = operand$
148.       End for
149.       $exp =$ new Expression($operator$, $ops$)
150.       Return $exp$
151.    End Function
```

# Appendix I

# XML and Ontologies

In this appendix, we give a background on the field of XML-to-ontology mapping. We start by an overview on XML technology and its related technologies, such as XML Schema. Then, we discuss about XML as data sources and about the contexts in which XML-to-ontology mapping is needed. We also give a brief comparison between XML and ontologies.

## I.1 What is XML ?

XML [33] is a markup language for documents containing structured information. XML is designed to carry data in a self-descriptive manner. It is a subset of the ISO standard SGML (Standard General Markup Language). SGML [92] was created to specify document markup languages or tag sets for describing electronic texts. XML development was started in 1996 by the XML Working Group of the World Wide Web Consortium (W3C), for ease of implementation and interoperability with both SGML and HTML. XML became a W3C Recommendation in February 1998.

XML was designed to overcome some of the drawbacks of HTML. For instance, HTML was meant to be consumed only by human readers since it dealt only with content presentation of Web resources, not with the structure of that content. Currently, XML is being used not only to structure texts but to exchange a wide variety of data on the Web, allowing better interoperability between information systems. XML allows users to define their own tags and attributes, define data structures (nesting them), and extract data from documents. As a language for the World Wide Web, the main advantages of XML are: 1) it is easy to parse, 2) its syntax is well defined, and 3) it is human readable.

The main structure of an XML document is tree-like. The main nodes in this tree model are Elements, Attributes and Text (Data). Element nodes can have children node that can be attributes, text, or other elements (called subelements). Attributes and text are leave nodes and have no children. Elements are the most basic information units. They are pieces of text or other nested XML elements enclosed in tags (one start tag and one end tag with the same name). Attributes are used to describe elements. They associate name-value pairs to elements and appear only in start tags or empty-element tags. Text are the parsed character data between two tags.

An XML document must be *well formed*, that is, it must have correct XML syntax. In a well-formed XML document only one root element exists, all its elements are correctly nested (the start and end tags are balanced), and attribute values are delimited by quotes (“).

One of the important innovations of XML is the ability to place preconditions on the data in a simple declarative way. Checking an XML document against this list of conditions is called validation. Validation is an optional step but an important one. A *valid* XML document is a *well formed* XML document whose structure is defined by a schema and is compliant with it (that is, it follows the structure described in it). There are two well-known languages that allow to express the structure of XML document, namely, DTD and XML Schema.

DTD is a standard for defining the legal elements in an XML document. A Document Type Definition (DTD) is a document that specifies constraints on the valid tags and tag sequences that can appear in an XML document. In other words, it defines the structure of an XML document with a list of legal elements and attributes.

In the reminder of this chapter and the next chapter, we will extensively focus on XML Schema, thus, we prefer to devote the next subsection to introduce this topic as a reminder for the reader.

## I.2 XML Schema

XML schemas [155][22] is a language to specify the structure and constraints of XML documents (including datatypes, cardinalities, minimum and maximum values, etc.). The XML Schema language is also referred to as XML Schema Definition (XSD). XML Schema became a W3C Recommendation in 2001.

XML schemas provide several significant improvements over DTDs:

- Definitions are itself XML documents. The clear advantage is that all tools developed for XML (e.g., validation or rendering tools) can be immediately applied to XML schema definitions.
- A rich set of datatypes that can be used to define the values of elementary tags.
- Much richer means for defining nested tags (i.e., tags with sub-tags).
- The namespace mechanism to combine XML documents with heterogeneous vocabulary.

An XML schema basically consists of element declarations, attribute declarations, simple and complex type definitions. These declarations describe the structure of an XML document, that means which elements, attributes and types are allowed or required as the content of an element and in which order.

Elements, types, and attributes can independently be locally or globally scoped. A global declaration is globally accessible in the schema. It is declared as a direct child of the ⟨`schema`⟩ tag. It can be reused/referenced by any element/attribute within the schema. A local declaration is declared as a non-direct descendant of the ⟨`schema`⟩ tag. It applies only to the contents of its enclosed element/type, and cannot be reused.

XML Schema recommendation provides a wide variety of predefined atomic datatypes. These datatypes are documented in [22], and are known as XSD datatypes. In addition, XML Schema allows users to create their custom data types. Custom data types can be simple and complex data types. A simple data type is a predefined datatype, such as string or an integer, but with some refinement. A simple element is an XML element that can contain only text. It cannot contain any other elements or attributes. A complex data type is a custom-built data type consisting of multiple elements and/or attributes. A complex element is an XML element that contains other elements and/or attributes.

XML Schema offers three compositors to combine elements: sequence, all and choice. A compositor is a number of elements, contained in some other collection definition. In xsd:sequence compositor, child elements must appear in a specific order. In xsd:all compositor, child elements can appear in any order, each child element must occur only once. In a xsd:choice compositor, either one child element or another can appear.

A simple type can be derived from an atomic type by defining restrictions on values, set of values, or a series of values of the base type. A complex type can be derived from a simple type, by extending it with additional attributes. A complex type can also be derived from another (base) complex type, either by extend the base type with additional contents (elements and/or attributes), or by restricting the base type using facets or cardinalities.

XML Schema language offers various options for how to organize an XML schema. Thus, several design patterns of XML schemas have arisen, such as: Russian Doll, Salami Slice, Venetian Blind and Garden of Eden. We think it is useful to present these design patterns in this introductive background, since we will need them in the next chapter. The following section is dedicated to present the most well known XML schema design patterns.

## I.3 XML Schema Design Patterns

XML Schema offers many options for schema organization. First of all, elements and types can be managed separately. This allows to reuse a type definition by multiple elements, or to reference an element declaration within multiple types. In addition, type inheritance and derivation allows for deep type hierarchies.

Elements, types, and attributes can independently be locally or globally scoped. This allows to customize the scope of each component within the schema as desired. Finally, namespace support allows for distributed component creation and reuse, and to import outer schemas that can reset some settings.

This variety of choices leads to arise several design options of XML schemas. The most well known design patterns of XML schemas are [117]: Russian Doll, Salami Slice, Venetian Blind and Garden of Eden. These patterns differs in particular according to the scope of declared elements and types. That is, when should an element or type be declared global versus when should it be declared local?.

### Russian Doll

The Russian Doll design contains only one single global element; all other elements are local. Element declarations are nested within a single global declaration and element declarations can only be used once. Only the root element must be defined within the global namespace.

Since it contains only one single global element, Russian Doll is the simplest and easiest pattern to use by instance developers. However, if its elements or types are intended for reuse, Russian Doll is not suitable for schema developers.

```
<xsd:element name="Line">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="PointA">
        <xsd:complexType>
          <xsd:attribute name="x" type="xsd:integer"/>
          <xsd:attribute name="y" type="xsd:integer"/>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="PointB">
        <xsd:complexType>
```

```
          <xsd:attribute name="x" type="xsd:integer"/>
          <xsd:attribute name="y" type="xsd:integer"/>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

## Salami Slice

In the Salami Slice design, all elements are global. There is no nesting of element declarations and element declarations can be reused throughout the schema. All elements must be defined within the global namespace.

The fact that all the elements in Salami Slice are global means a greater degree of reusability than Russian Doll and Venetian Blind. However, this design pattern contains many potential root elements.

```
<xsd:element name="PointA">
  <xsd:complexType>
    <xsd:attribute name="x" type="xsd:integer"/>
    <xsd:attribute name="y" type="xsd:integer"/>
  </xsd:complexType>
</xsd:element>
<xsd:element name="PointB">
  <xsd:complexType>
    <xsd:attribute name="x" type="xsd:integer"/>
    <xsd:attribute name="y" type="xsd:integer"/>
  </xsd:complexType>
</xsd:element>
<xsd:element name="Line">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="PointA"/>
      <xsd:element ref="PointB"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

## Venetian Blind

The Venetian Blind design contains only one global element. All the other elements are local. Element declarations are nested within a single global declaration, using named complex types and element groups. Complex types and element groups can be reused throughout the schema. Only the root element must be defined within the global namespace.

Venetian Blind is an extension of Russian Doll, in which all the types are defined globally. Because it has only one single root element and all its types are reusable, Venetian Blind is suitable for use by both instance developers and schema developers.

```
<xsd:complexType name="PointType">
  <xsd:attribute name="x" type="xsd:integer"/>
  <xsd:attribute name="y" type="xsd:integer"/>
</xsd:complexType>
<xsd:element name="Line">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="PointA" type="PointType"/>
      <xsd:element name="PointB" type="PointType"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

### Garden of Eden

The Garden of Eden design is a combination of the Venetian Blind and Salami Slice designs. All elements and types are defined in the global namespace with the elements referenced as needed.

Because it exposes all its elements and types globally, Garden of Eden, like Salami Slice, is completely reusable. However, because Garden of Eden exposes multiple elements as global ones, there are many potential root elements.

```
<xsd:complexType name="PointType">
  <xsd:attribute name="x" type="xsd:integer"/>
  <xsd:attribute name="y" type="xsd:integer"/>
</xsd:complexType>
<xsd:complexType name="LineType">
  <xsd:sequence>
    <xsd:element ref="PointA"/>
    <xsd:element ref="PointB"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:element name="PointA" type="PointType"/>
<xsd:element name="PointB" type="PointType"/>
<xsd:element name="Line" type="LineType"/>
```

In conclusion, XML Schema language offers a variety of design choices that leads to arise several design styles/patterns. We will see in the next chapter that our XML-to-ontology mapping tool (called X2OWL) works for all of these XML design patterns. In fact, if the same XML data source is designed using different patterns (giving different XML schemas), then the ontologies generated from these schemas (using X2OWL) will hold the same semantics, we will obtain the same ontology from these schemas.

Now, as we introduced XML and XML schema, we are going to investigate the field of XML-to-ontology mapping. We start by discussing about XML as means to store information (Section I.4), and we make a brief comparison between XML and ontologies (Section I.5). Then, we see the contexts within which XML data sources need to be mapped to ontologies (Section I.6).

## I.4 XML as Data Sources

Today, XML has reached a wide acceptance as data exchange format. The increasingly common use of XML for data transport has encouraged enterprises to use XML databases to store their data assets in XML format. Consequently, efficient XML document storage has become a core data management issue, and several XML storage approaches have emerged such as XML-enabled and Native XML databases.

An XML database [135] is a data persistence software system that allows data to be stored in XML format. This data can then be queried, exported and serialized into the desired format. The main reason for the use of XML in databases: the increasingly common use of XML for data transport, which means that "data is extracted from databases and put into XML documents and vice-versa". It may prove more efficient (in terms of conversion costs) and easier to store the data in XML format.

Two major classes of XML database exist:

1. **XML Enabled Database (XEDB)** – In this class of databases, all XML data is mapped to a traditional database (such as a relational database), accepting XML as input and rendering XML as output. This term implies that the database does the conversion itself (as opposed to relying on middleware). Examples of this class of databases are: Oracle XML DB[1] and SQL Server 2005[2]
2. **Native XML Database (NXD)** – Such databases define a (logical) model for an XML document and stores and retrieves documents according to that model. This model uses XML documents as the fundamental unit of (logical) storage, however, it is not required to have any particular underlying physical storage model. Examples of native XML database systems are: XStreamDB[3] and Tamino XML Server[4]

However, independently of whether the data is actually stored in XML native mode or in an XML enabled database, the view, presented to the users, is XML-based. Thus, the use of XML as a data representation and exchange standard raises new issues for data management [154].

In the following sections, we will use the term XML data source to denote either:

1. an XML document with no XML schema, or
2. one or more XML documents that are compliant to one XML schema.

In the next section, we give a brief comparison between XML and ontologies.

## I.5 XML and Ontologies

In Section B.2, we made a comparison between databases and ontologies in order to introduce database-to-ontology mapping issues and approaches. Here also, we compare XML/XML

[1]http://www.oracle.com/technology/tech/xml/xmldb/index.html
[2]http://msdn.microsoft.com/en-us/library/ms345117(SQL.90).aspx
[3]http://www.bluestream.com/products/content/bluestream-xstreamdb/
[4]http://www.softwareag.com/corporate/products/wm/tamino/

Schema and ontologies, since such a comparison serves as an introduction to XML-to-ontology mapping approaches. This comparison is given at two levels: data level (that is, XML versus RDF), and schema level (that is, XML Schema and ontology schemas).

**XML versus RDF**

XML is a language that defines a generic syntax to store and exchange documents by means of a tree-based structure. Although RDF has an XML-based syntax, XML and RDF serve different purposes and have been developed separately within the W3C. Thus, they have different modeling foundations. XML is based on a tree model where only nodes are labeled and the outgoing edges are ordered. In contrast to this, RDF is based on a directed graph model where edges have labels but are unordered. It distinguishes between resources (e.g. car) and properties (e.g. car color) while XML does not (e.g. both would be elements).

**XML Schema versus Ontology Languages**

The differences between XML schema and ontology are generally divided into three groups: data type, structure and relation [171].

### 1. Data type

XML schema supports large number of built-in data types including string, boolean, decimal, float, date, etc. A complete list of datatypes is documented in [22].

Some ontology languages such as RDF and OIL only supports limited number of data types. Others such as DAML and OWL allows use of XML Schema datatypes by referring to the datatype URI.

### 2. Structure

XML schema uses nested data structure, where each element can be mixed with other simple, complex or mixed elements. The top-most element is considered as the root element of the concept hierarchy. Upper elements are seen as the parent of its content lower elements.

Ontology supports element composition through properties. Each class can have various datatype properties and object properties. However, ontology is an object-oriented conceptual model rather than a hierarchy of terms or concepts. Therefore, every class existing in the ontology can be seen as the root element.

XML schema allows the definition of structural constraints, concepts such as sequence is used to describe the order between content items. Ontology does not support order between properties.

### 3. Relation

XML schema only supports inheritance through type derivation (extension or restriction). It does not support multiple-inheritance. Ontology supports multiple-inheritance, one class can inherit properties from multiple parent classes.

XML schema does not provide grammars for relation constraints definition. Ontology supports inheritance on properties, it also provides simple logics on relations such as transitive and symmetric for reasoning on class.

## I.6 Why to Map XML to Ontologies?

In Section 5.1.1, we saw that in some contexts, there is a need to map databases to ontologies. Those contexts are: ontology-based information integration and the semantic web. In these same contexts, when the underlying data are stored in XML format, there is also a need to map XML to ontologies.

As we mentioned, currently we witness an important tendency towards XML-based data storage and the emergence of XML databases. In the context of ontology-based information integration, ontologies are usually used for the explication of implicit and hidden knowledge in information sources involved in the integration. In this context, when information are stored in XML data sources, the XML-to-ontology mappings are essential to relate the ontologies to the actual content of information sources.

With the current emergence of the Semantic Web, we also observe a tendency towards moving existing XML data to the Semantic Web. The motivation of XML-to-ontology mapping, is basically to enrich the pure syntactic nature of XML, and move XML to semantically richer nature provided by ontologies.

In the literature, several approaches have been proposed to relate ontologies with XML data sources. In this chapter, we make a literature review to investigate some of existing approaches for XML-to-ontology mapping. In general, these approaches can be classified into two main categories:

1. Approaches that create an ontology from XML document (Section 9.1).
2. Approaches that map an XML document to an existing ontology (Section 9.2).

A survey on these approaches is given in Chapter 9.

Appendix J

# Algorithms for ontology generation in X2OWL tool

This appendix contains the complete algorithms mentioned in Chapter 10 for ontology generation from an XML data source. These algorithmes are:

- Generate classes algorithm.
- Check root vertex algorithm.
- Check vertex algorithm.

## Generate Classes Algorithm

Table J.1: X2OWL: Generate Classes Algorithm

```
ALGORITHM: generateClasses
INPUT: $XSG = (V, E)$, $derivedTypesMap$
BEGIN
    $classMap$ = new Map
    FOR each $v \in V$
        IF $v$ is complex type vertex
            LET $CT$ be the complex type of $v$
            IF $CT$ is global
                $C$ = createClass($CT$.getName())
                $classMap$.put($v$, $C$)
            ELSE /* $CT$ is local, so anonymous */
                LET $e$ be the (only) incoming edge of $v$
                $v_{source} = e$.getSource()
                LET $el_{source}$ be the element of $v_{source}$
                $C$ = createClass($el_{source}$.getName())
                $classMap$.put($v$, $C$)
            END IF
        ELSE IF $v$ is element group vertex
            LET $EG$ be the element group of $v$
            $C$ = createClass($EG$.getName())
            $classMap$.put($v$, $C$)
        ELSE IF $v$ is attribute group vertex
            LET $AG$ be the element group of $v$
            $C$ = createClass($AG$.getName())
            $classMap$.put($v$, $C$)
        END IF
    END FOR

    /* setup the class hierarchy */
    FOR each $entry \in derivedTypesMap$
        $v_{derivedType} = entry$.getKey()
        $v_{baseType} = entry$.getvalue()
```

```
33.             $C_{child} = classMap$.get($v_{derivedType}$)
34.             $C_{parent} = classMap$.get($v_{baseType}$)
35.             $C_{parent}$.setSubClass($C_{child}$)
36.         END FOR
37.     END
```

## Check Root Vertex Algorithm

```
1.  ALGORITHM: checkRootVertex
2.  INPUT: root vertex $v_{root}$
3.  BEGIN
4.      LET $el_{root}$ be the element of $v_{root}$
5.      $xpath_{root}$ = "/" + $el_{root}$.getName()
6.      LET $CT_{root}$ be the complex type of $el_{root}$
7.      LET $e$ be the (only) outgoing edge of $v_{root}$
8.      $v_{ct} = e$.getDest() /* vertex corresponding to CTroot */
9.      $C_{root} = classMap$.get($v_{ct}$) /* class corresponding to CTroot */
10.     $CB_{root}$ = createConceptBridge($C_{root}$, $xpath_{root}$)
11.     FOR each child $v_{child}$ of $v$
12.         checkVertex($v_{child}$, $xpath_{root}$, $C_{root}$, $CB_{root}$)
13.     END FOR
14. END
```

## Check Vertex Algorithm

```
1.  ALGORITHM: checkVertex
2.  INPUT: $v$, $xpath_{parent}$ , $C_{parent}$ , $CB_{parent}$.
3.  BEGIN
4.      IF $v$ is element vertex
5.          LET $el$ be the element of $v$
6.          $xpath_{el}$ = $xpath_{parent}$ + "/" + $el$.getName()
7.          IF $el$ has a complex type
8.              LET $CT_{el}$ be the complex type of $el$
9.              LET $e$ be the outgoing edge of $v$
10.             $v_{ct} = e$.getDest() /* vertex corresponding to $CT_{el}$ */
11.             $C_{el} = classMap$.get($v_{ct}$)
12.             $CB_{el}$ = CB($C_{el}$, $xpath_{el}$)
13.             $OP_{el}$ = createObjectProperty("has"+$el$.getName(),$C_{parent}$,$C_{el}$)
14.             $OPB_{el}$ = OPB($OP_{el}$,$CB_{parent}$,$CB_{el}$)
15.             FOR each child $v_{child}$ of $v$
```

```
16.                 checkVertex($v_{child}$, $xpath_{el}$, $C_{el}$, $CB_{el}$)
17.             END FOR
18.         ELSE IF $el$ has a simple type $ST_{el}$
19.             $DP_{el}$ = createDatatypeProperty($el$.getName(),$C_{parent}$,$ST_{el}$)
20.             $DPB_{el}$ = DPB($DP_{el}$,$CB_{parent}$,$xpath_{el}$+"/text()")
21.         END IF
22.     ELSE IF $v$ is attribute vertex
23.         LET $att$ be the attribute of $v$
24.         $xpath_{att} = xpath_{parent}+$ "/@" + $att$.getName()
25.         LET $ST_{att}$ be the (simple) type of $att$
26.         $DP_{att}$ = createDatatypeProperty($att$.getName(), $C_{parent}$, $ST_{att}$)
27.         $DPB_{att}$ = DPB($DP_{att}$, $CB_{parent}$, $xpath_{att}$)
28.     ELSE IF $v$ is complex type vertex
29.         FOR each child $v_{child}$ of $v$
30.             checkVertex($v_{child}$, $xpath_{parent}$, $C_{parent}$, $CB_{parent}$)
31.         END FOR
32.     ELSE IF $v$ is element group vertex
33.         LET $eg$ be the element group of $v$
34.         $C_{eg} = classMap$.get($v$)
35.         $CB_{eg}$ = CB($C_{eg}$, $xpath_{parent}$)
36.         $OP_{eg}$ = createObjectProperty($eg$.getName(), $C_{parent}$, $C_{eg}$)
37.         $OPB_{eg}$ = OPB($OP_{eg}$, $CB_{parent}$, $CB_{eg}$)
38.         FOR each child $v_{child}$ of $v$
39.             checkVertex($v_{child}$, $xpath_{parent}$, $C_{eg}$, $CB_{eg}$)
40.         END FOR
41.     ELSE IF $v$ is attribute group vertex
42.         LET $ag$ be the attribute group of $v$
43.         $C_{ag} = classMap$.get($v$)
44.         $CB_{ag}$ = CB($C_{ag}$, $xpath_{parent}$)
45.         $OP_{ag}$ = createObjectProperty($ag$.getName(), $C_{parent}$, $C_{ag}$)
46.         $OPB_{ag}$ = OPB($OP_{ag}$, $CB_{parent}$, $CB_{ag}$)
47.         FOR each child $v_{child}$ of $v$
48.             checkVertex($v_{child}$, $xpath_{parent}$, $C_{ag}$, $CB_{ag}$)
49.         END FOR
50.     END IF
51. END
```

# Appendix K

# XOML Document for Running Example of Chapters 10 and 11

```
@prefix ex: <http://www.something.com/2009/myontology#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xoml: <http://www.xoml.org/mapping-specification#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix map: <http://www.something.com/2009/my-xoml-map#> .
@prefix daml: <http://www.daml.org/2001/03/daml+oil#> .

# concept bridges description

map:cb1    a      xoml:ConceptBridge ;
    xoml:class    ex:shiporder ;
    xoml:xpath    "/shiporder" .

map:cb2    a      xoml:ConceptBridge ;
    xoml:class    ex:ships ;
    xoml:xpath    "/shiporder/ships" .

map:cb3    a      xoml:ConceptBridge ;
    xoml:class    ex:ship ;
    xoml:xpath    "/shiporder/ships/ship" .

map:cb4    a      xoml:ConceptBridge ;
    xoml:class    ex:item ;
    xoml:xpath    "/shiporder/ships/ship/item" .

map:cb5    a      xoml:ConceptBridge ;
    xoml:class    ex:items ;
    xoml:xpath    "/shiporder/items" .

map:cb6    a      xoml:ConceptBridge ;
    xoml:class    ex:item ;
    xoml:xpath    "/shiporder/items/item" .

# datatype property bridges description

map:dpb1    a    xoml:DatatypePropertyBridge ;
    xoml:belongsToConceptBridge    map:cb1 ;
    xoml:datatypeProperty          ex:orderid ;
    xoml:xpath    "/shiporder/@orderid" .

map:dpb2    a    xoml:DatatypePropertyBridge ;
    xoml:belongsToConceptBridge    map:cb1 ;
    xoml:datatypeProperty          ex:orderperson ;
    xoml:xpath    "/shiporder/orderperson/text()" .

map:dpb3    a    xoml:DatatypePropertyBridge ;
    xoml:belongsToConceptBridge    map:cb3 ;
    xoml:datatypeProperty          ex:date ;
    xoml:xpath    "/shiporder/ships/ship/date/text()" .
```

```

map:dpb4    a    xoml:DatatypePropertyBridge ;
    xoml:belongsToConceptBridge    map:cb4 ;
    xoml:datatypeProperty          ex:title ;
    xoml:xpath    "/shiporder/ships/ship/item/@title" .

map:dpb9    a    xoml:DatatypePropertyBridge ;
    xoml:belongsToConceptBridge    map:cb6 ;
    xoml:datatypeProperty          ex:quantity ;
    xoml:xpath    "/shiporder/items/item/quantity/text()" .

map:dpb10    a    xoml:DatatypePropertyBridge ;
    xoml:belongsToConceptBridge    map:cb6 ;
    xoml:datatypeProperty          ex:title ;
    xoml:xpath    "/shiporder/items/item/title/text()" .

map:dpb11    a    xoml:DatatypePropertyBridge ;
    xoml:belongsToConceptBridge    map:cb6 ;
    xoml:datatypeProperty          ex:price ;
    xoml:xpath    "/shiporder/items/item/price/text()" .

# object property bridges description

map:opb1    a    xoml:ObjectPropertyBridge ;
    xoml:belongsToConceptBridge    map:cb1 ;
    xoml:objectProperty            ex:shiporder-ships ;
    xoml:refersToConceptBridge     map:cb2 .

map:opb2    a    xoml:ObjectPropertyBridge ;
    xoml:belongsToConceptBridge    map:cb2 ;
    xoml:objectProperty            ex:ships-ship ;
    xoml:refersToConceptBridge     map:cb3 .

map:opb3    a    xoml:ObjectPropertyBridge ;
    xoml:belongsToConceptBridge    map:cb3 ;
    xoml:objectProperty            ex:ship-item ;
    xoml:refersToConceptBridge     map:cb4 .

map:opb4    a    xoml:ObjectPropertyBridge ;
    xoml:belongsToConceptBridge    map:cb1 ;
    xoml:objectProperty            ex:shiporder-items ;
    xoml:refersToConceptBridge     map:cb5 .

map:opb5    a    xoml:ObjectPropertyBridge ;
    xoml:belongsToConceptBridge    map:cb5 ;
    xoml:objectProperty            ex:items-item ;
    xoml:refersToConceptBridge     map:cb6 .

```

Appendix L

# Algorithms for SPARQL-to-XQuery Translation

This appendix contains the complete algorithms mentioned in our SPARQL-to-XQuery Translation method (Chapter 11). These algorithmes are:

- Find-Pairs Algorithm.
- Find-Mapping-Graphs Algorithm.

## L.1 Find-Pairs Algorithm

```
ALGORITHM: FIND-PAIRS
INPUT: a triple $t \in T$, and a XOML parser $xomlParser$
OUTPUT: a set of pairs $P$
BEGIN
    $P = \phi$
    $prdNode$ = $t$.getPredicate()
    $subNode$ = $t$.getSubject()
    $objNode$ = $t$.getObject()
    IF $subNode$ is variable THEN $subVar$ = $subNode$.asVariable()
    IF $objNode$ is variable THEN $objVar$ = $objNode$.asVariable()
    IF $prdNode$ is a datatype property
        $DPBs$ = $xomlParser$.getDPBs($prdNode$)
        FOR each $dpb \in DPBs$
            $cb$ = $dpb$.getDomCB()
            $xpath_{cb}$ = $cb$.getXpath()
            $vx_{from}$ = new VAR2XPATH($subVar$, $xpath_{cb}$)
            $xpath_{dpb}$ = $dpb$.getXPath()
            IF $objNode$ is variable
                $vx_{to}$ = new VAR2XPATH($objVar$, $xpath_{dpb}$)
            ELSE IF $objNode$ is literal
                $vx_{to}$ = new LITERAL2XPATH($objNode$.asLiteral(), $xpath_{dpb}$)
            END IF
            $pair$ = new Pair($vx_{from}$, $vx_{to}$)
            $P+ = pair$
        END FOR
    ELSE IF $prdNode$ is an object property
        $OPBs$ = $xomlParser$.getOPBs($prdNode$)
        FOR each $opb \in OPBs$
            $cb^{dom}$ = $opb$.getDomCB()
            $xpath_{cb^{dom}}$ = $cb^{dom}$.getXPath()
            $vx_{from}$ = new VAR2XPATH($subVar$, $xpath_{cb^{dom}}$)
            IF $objNode$ is variable
                $cb^{rng}$ = $opb$.getRngCB()
                $xpath_{cb^{rng}}$ = $cb^{rng}$.getXpath()
```

```
35.             $vx_{to}$ = new VAR2XPATH($objVar$, $xpath_{cb^{rng}}$)
36.             $pair$ = new Pair($vx_{from}$, $vx_{to}$)
37.             $P+ = pair$
38.          END IF
39.       END FOR
40.    END IF
41. RETURN $P$
42. END
```

## L.2 Find-Mapping-Graphs Algorithm

```
1.  ALGORITHM: FIND-MAPPING-GRAPHS
2.  INPUT: a set of triples T
3.  OUTPUT: a set of mapping graphs MG
4.  BEGIN
5.     $MG = \phi$
6.     $pairsMap$ = new Map()
7.     FOR each triple $t \in T$
8.        $P^t$ = FIND-PAIRS($t$)
9.        pairsMap.put($t$,$P^t$)
10.    END FOR
11.    $n$ = $T$.size()
12.    $mg_0$ = new MappingGraph($n$)
13.    $MG+ = mg_0$
14.    FOR $i = 1$ TO $n$
15.       $t_i = T[i]$
16.       $P_i$ = $pairsMap$.get($t_i$)
17.       $MG_{new} = \phi$
18.       FOR $k = 1$ TO $MG$.size()
19.          $mg = MG[k]$
20.          FOR $j = 1$ TO $P_i$.size()
21.             $mg_{new}$ = $mg$.copy()
22.             $mg_{new}$.setPair($P_i[j]$,$i$)
23.             $MG_{new}+ = mg_{new}$
24.          END FOR
25.       END FOR
26.       $MG = MG_{new}$
27.    END FOR
28.    RETURN $MG$
29. END
```

# Bibliography

[1] Abrouk, L., Cullot, N., Ghawi, R., Gomez-Carpio, G. V., and Poulain, T. Cooperation of Information Sources in OWSCIS System. In *Collaborative and Grid Computing Technologies (CGCT2008)* (2008).

[2] Alexiev, V., Breu, M., Bruijn, J. d., Fensel, D., Lara, R., and Lausen, H. *Information Integration with Ontologies: Experiences from an Industrial Showcase.* John Wiley & Sons, 2005.

[3] An, Y., Mylopoulos, J., and Borgida, A. Building Semantic Mappings from Databases to Ontologies. In *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference (AAAI)* (July 2006), AAAI Press.

[4] Andersson, M. Extracting an Entity Relationship Schema from a Relational Database through Reverse Engineering. In *ER '94: Proceedings of the13th International Conference on the Entity-Relationship Approach* (London, UK, 1994), Springer-Verlag, pp. 403–419.

[5] Anicic, N., Ivezic, N., and Marjanovic, Z. Mapping XML Schema to OWL. In *Enterprise Interoperability* (2007).

[6] Atzeni, P., and Torlone, R. MDM: a Multiple-Data-Model Tool for the Management of Heterogeneous Database Schemes. In *Proceedings of ACM SIGMOD, Exhibition Section* (1997), pp. 528–531.

[7] Baader, F., Horrocks, I., and Sattler, U. Description Logics as Ontology Languages for the Semantic Web. In *Festschrift in honor of Jrg Siekmann, Lecture Notes in Artificial Intelligence* (2003), Springer-Verlag, pp. 228–248.

[8] Barrasa-Rodriguez, J., Corcho, O., and Gomez-Perez, A. R2O, an Extensible and Semantically based Database-to-Ontology Mapping Language. In *2nd Workshop on Semantic Web and Databases (SWDB2004)* (2004).

[9] Barrasa-Rodriguez, J., and Gomez-Perez, A. Upgrading Relational Legacy Data to the Semantic Web. In *Proceedings of the 15th international conference on World Wide Web (WWW '06)* (New York, NY, USA, 2006), ACM, pp. 1069–1070.

[10] Barsalou, T., and Gangopadhyay, D. M(DM): An Open Framework for Interoperation of Multimodel Multidatabase Systems. In *Proceedings of the Eighth International*

*Conference on Data Engineering, February 3-7, 1992, Tempe, Arizona* (1992), F. Golshani, Ed., IEEE Computer Society, pp. 218–227.

[11] Bayardo, Jr., R. J., Bohrer, W., Brice, R., Cichocki, A., Fowler, J., Helal, A., Kashyap, V., Ksiezyk, T., Martin, G., Nodine, M., Rashid, M., Rusinkiewicz, M., Shea, R., Unnikrishnan, C., Unruh, A., and Woelk, D. InfoSleuth: Agent-based Semantic Integration of Information in Open and Dynamic Environments. In *Proceedings ACM SIGMOD International Conference on Management of Data, SIGMOD '97* (New York, NY, USA, 1997), vol. 26, ACM, pp. 195–206.

[12] Beckett, D., and Broekstra, J. SPARQL Query Results XML Format. Tech. rep., W3C, April 2006.

[13] Beckett, D., and Grant, J. SWAD-Europe Deliverable 10.2: Mapping Semantic Web Data with RDBMSes. Tech. rep., W3C, 2003.

[14] Belkin, N. J., and Croft, W. B. Information Filtering and Information Retrieval: Two Sides of the Same Coin? *Communications of the ACM 35*, 12 (1992), 29–38.

[15] Benjamins, R. V., Fensel, D., Decker, S., and Gomez-Perez, A. (KA)2: Building Ontologies for the Internet: a Mid Term Report. *International Journal of Human-Computer Studies 51*, 3 (September 1999), 687–712.

[16] Bergamaschi, S., Castano, S., and Vincini, M. Semantic Integration of Semistructured and Structured Data Sources. *SIGMOD Rec. 28*, 1 (1999), 54–59.

[17] Bergamaschi, S., Castano, S., Vincini, M., and Beneventano, D. Semantic Integration of Heterogeneous Information Sources. *Data and Knowledge Engineering 36*, 3 (2001), 215–249.

[18] Bergamaschi, S., Sartori, C., Beneventano, D., and Vincini, M. ODB-Tools: A Description Logics based Tool for Schema Validation and Semantic Query Optimization in Object Oriented Databases. In *AI*IA 97: Advances in Artificial Intelligence, 5th Congress of the Italian Association for Artificial Intelligence, Rome, Italy* (1997), vol. 1321 of *Lecture Notes in Computer Science*, Springer, Berlin, pp. 435–438.

[19] Bergman, M. K. The Deep Web: Surfacing Hidden Value. *Journal of Electronic Publishing 7*, 1 (August 2001).

[20] Berners-Lee, T. Relational Databases on the Semantic Web, September 1998. Design Issues (published on the Web).

[21] Berners-Lee, T., Hendler, J., and Lassila, O. The Semantic Web. *Scientific American Magazine 1* (May 2001), 29–37.

[22] Biron, P. V., and Malhotra, A. XML Schema Part 2: Datatypes Second Edition. Tech. rep., W3C, October 2004. W3C Recommendation.

[23] Bizer, C., Cyganiak, R., and Heath, T. How to Publish Linked Data on the Web, 2007.

[24] Bizer, C., Heath, T., Idehen, K., and Berners-Lee, T. Linked Data on the Web. In *Proceedings of the 17th International Conference on World Wide Web (WWW 2008)* (Beijing, China, April 2008), ACM, pp. 1265–1266.

[25] Bizer, C., and Seaborne, A. D2RQ - Treating Non-RDF Databases as Virtual RDF Graphs. In *3rd International Web Semantic Conference (ISWC 2004) (posters)* (November 2004).

[26] Blaha, M., Premerlani, W., and Shen, H. Converting OO Models into RDBMS Schema. *IEEE Software 11*, 3 (1994), 28–39.

[27] Bohring, H., and Auer, S. Mapping XML to OWL Ontologies. In *Leipziger Informatik-Tage, volume 72 of LNI* (2005), GI, pp. 147–156.

[28] Borgida, A., Brachman, R. J., McGuinness, D. L., and Resnick, L. A. CLASSIC: A Structural Data Model for Objects. In *SIGMOD '89: Proceedings of the 1989 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 1989), ACM, pp. 58–67.

[29] Borst, W. N. *Construction of Engineering Ontologies for Knowledge Sharing and Reuse.* PhD thesis, University of Tweenty, Enschede, The Netherlands, 1997.

[30] Bowers, S., and Delcambre, L. Representing and Transforming Model-Based Information. In *Proceedings of the Workshop on Semantic Web at ECDL-00* (Lisbon, Portugal, 2000).

[31] Boyd, M., Kittivoravitkul, S., Lazanitis, C., Brien, P. M. C., and Rizopoulos, N. AutoMed: A BAV Data Integration System for Heterogeneous Data Sources. In *Proceedings of 16th International Conference on Advanced Information Systems Engineering (CAiSE 2004), Riga, Latvia* (2004), vol. 3084 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 82–97.

[32] Brachman, R. J., and Schmolze, J. G. An Overview of the KL-ONE Knowledge Representation System. *Cognitive Science 9*, 2 (1985), 171–216.

[33] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and Yergeau, F. Extensible Markup Language (XML) 1.0 (Fifth Edition). W3c recommendation 26 november 2008, W3C, November 2008.

[34] Bressan, S., Goh, C., Levina, N., Madnick, S., Shah, A., and Siegel, M. Context Knowledge Representation and Reasoning in the Context Interchange System. *Applied Intelligence 13*, 2 (2000), 165–180.

[35] Brickley, D., and Guha, R. V. RDF Vocabulary Description Language 1.0: RDF Schema. Tech. rep., W3C, February 2004. W3C Recommendation.

[36] Broekstra, J., Kampman, A., and van Harmelen, F. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *Proceedings of the First International Semantic Web Conference (ISWC'02)* (July 2002), I. Horrocks and J. Hendler, Eds., no. 2342 in Lecture Notes in Computer Science, Springer Verlag, pp. 54–68.

[37] Buccella, A., Cechich, A., and Brisaboa, N. R. An Ontology Approach to Data Integration. *Journal of Computer Science and Technology 3* (2003), 62–68.

[38] Buccella, A., Cechich, A., and Brisaboa, N. R. Ontology-Based Data Integration Methods: A Framework for Comparison, 2005. Colombian Journal of Computation. v.6, n.1, 2005.

[39] Busse, S., Kutsche, R. D., Leser, U., and Weber, H. Federated Information Systems: Concepts, Terminology and Architectures. Tech. Rep. 99-9, TU Berlin, 1999.

[40] Carroll, J. J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., and Wilkinson, K. Jena: Implementing the Semantic Web Recommendations. In *Proceedings of the 13th international conference on World Wide Web - Alternate Track Papers & Posters, (WWW 2004)* (May 2004), pp. 74–83. New York, USA.

[41] Chaudhri, V. K., Farquhar, A., Fikes, R., Karp, P. D., and Rice, J. Open Knowledge Base Connectivity 2.0. Tech. Rep. KSL-98-06, Knowledge Systems, AI Laboratory, Stanford, CA, USA, January 1998.

[42] Chebotko, A., Lu, S., Jamil, H. M., and Fotouhi, F. Semantics Preserving SPARQL-to-SQL Query Translation for Optional Graph Patterns. Tech. Rep. TR-DB-052006-CLJF, Wayne State University, May 2006.

[43] Chen, P. P. The Entity-Relationship Model - Toward a Unified View of Data. *ACM Transactions on Database Systems 1*, 1 (1976), 9–36.

[44] Clark, K. G., Feigenbaum, L., and Torres, E. SPARQL Protocol for RDF. World Wide Web Consortium, Recommendation, January 2008.

[45] Cluet, S., Delobel, C., Simeon, J., and Smaga, K. Your Mediator Need Data Conversion! In *ACM SIGMOD International Conference on Management of Data* (June 1998), pp. 177–188.

[46] Codd, E. F. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM 13*, 6 (1970), 377–387.

[47] Corcho, O., and Gomez-Perez, A. Evaluating Knowledge Representation and Reasoning Capabilities of Ontology Specification Languages. In *Proceedings of the ECAI 2000 Workshop on Applications of Ontologies and Problem-Solving Methods, Berlin, 2000.* (2000).

[48] Cruz, I. F., Xiao, H., and Hsu, F. An Ontology-Based Framework for XML Semantic Integration. In *IDEAS '04: Proceedings of the International Database Engineering and Applications Symposium* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 217–226.

[49] Cui, Z., Jones, D., and O'Brien, P. Issues in Ontology-based Information Integration. In *In: Proceedings of Workshop on Ontologies and Information Sharing at the International Joint Conference on Artificial Intelligence 2001.* (Washington, USA, August 2001), Morgan Kaufmann Publishers: San Francisco, USA.

[50] Cui, Z., and O'Brien, P. Domain Ontology Management Environment. In *HICSS '00: Proceedings of the 33rd Hawaii International Conference on System Sciences-Volume 8* (Washington, DC, USA, 2000), IEEE Computer Society, p. 8015.

[51] Cullot, N., Ghawi, R., and Yetongnon, K. DB2OWL : A Tool for Automatic Database-to-Ontology Mapping. In *Proceedings of the Fifteenth Italian Symposium on Advanced Database Systems (SEBD 2007)* (Torre Canne, Fasano, BR, Italy, June 2007), pp. 491–494.

[52] Cyganiak, R. A Relational Algebra for SPARQL. Tech. Rep. HPL-2005-170, Digital Media Systems Laboratory - HP Laboratories Bristol, 2005.

[53] de Laborda, C. P., and Conrad, S. Relational.OWL - A Data and Schema Representation Format Based on OWL. In *Second Asia-Pacific Conference on Conceptual Modelling (APCCM2005)* (Newcastle, Australia, 2005), S. Hartmann and M. Stumptner, Eds., vol. 43 of *CRPIT*, ACS, pp. 89–96.

[54] de Laborda, C. P., and Conrad, S. Bringing Relational Data into the Semantic Web using SPARQL and Relational.OWL. In *ICDEW '06: Proceedings of the 22nd International Conference on Data Engineering Workshops* (Washington, DC, USA, 2006), IEEE Computer Society, p. 55.

[55] Deschaine, L. M., Brice, R. S., and Nodine, M. H. Use of InfoSleuth to Coordinate Information Acquisition, Tracking and Analysis in Complex Applications. Tech. Rep. INSL-008-00, Microelectronics and Computer Technology Corporation (MCC), 2000.

[56] Domingue, J. Tadzebao and WebOnto: Discussing, Browsing, and Editing Ontologies on the Web. In *Proceedings of the 11th Workshop on Knowledge Acquisition, Modeling and Management (KAW'98). Banff, Canada* (1998).

[57] Domingue, J., Motta, E., and Watt, S. The Emerging VITAL Workbench. In *Proceedings of the 7th European Workshop on Knowledge Acquisition for Knowledge-Based Systems* (London, UK, 1993), Springer-Verlag, pp. 320–339.

[58] Duschka, O. M., and Genesereth, M. R. Infomaster: An Information Integration Tool. In *In Proceedings of the International Workshop Intelligent Information Integration during the 21st German Annual Conference on Artificial Intelligence* (1997), pp. 9–12.

[59] Erl, T. *Service-Oriented Architecture: Concepts, Technology, and Design.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[60] Farquhar, A., Fikes, R., and Rice, J. The Ontolingua Server: a Tool for Collaborative Ontology Construction. In *International Journal of Human-Computer Studies* (1997), vol. 46(6), pp. 707–727.

[61] Fellbaum, C., Ed. *WordNet: An Electronic Lexical Database.* MIT Press, Cambridge, MA ; London, May 1998.

[62] Fensel, D., Horrocks, I., Harmelen, F. v., Decker, S., Erdmann, M., and Klein, M. C. A. OIL in a Nutshell. In *EKAW '00: Proceedings of the 12th European Workshop on Knowledge Acquisition, Modeling and Management* (London, UK, 2000), Springer-Verlag, pp. 1–16.

[63] Ferdinand, M., Zirpins, C., and Trastour, D. Lifting XML Schema to OWL. In *Web Engineering - 4th International Conference, ICWE 2004, Munich, Germany, July 26-30, 2004, Proceedings* (2004), N. Koch, P. Fraternali, and M. Wirsing, Eds., Springer Heidelberg, pp. 354–358.

[64] Fikes, R., Hayes, P., and Horrocks, I. OWL-QL - A Language for Deductive Query Answering on the Semantic Web. *Journal of Web Semantics 2*, 1 (2005).

[65] Fikes, R., Hayes, P. J., and Horrocks, I. DQL - A Query Language for the Semantic Web. Tech. Rep. KSL-02-05, Knowledge Systems, AI Laboratory, 2002.

[66] Firat, A., Madnick, S., and Grosof, B. Knowledge Integration to Overcome Ontological Heterogeneity: Challenges from Financial Information Systems. In *Proceedings of Twenty-Third International Conference on Information Systems ICIS-2002, Barcelona, Spain* (2002), pp. 183–194.

[67] Fowler, J., Perry, B., Nodine, M., and Bargmeyer, B. Agent-based Semantic Interoperability in InfoSleuth. *ACM SIGMOD Record 28*, 1 (1999), 60–67.

[68] Friedman, M., Levy, A., and Millstein, T. Navigational Plans for Data Integration. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)* (1999), American Association for Artificial Intelligence (AAAI), pp. 67–73.

[69] Garcia-Molina, H., Hammer, J., Ireland, K., Papakonstantinou, Y., Ullman, J., and Widom, J. Integrating and Accessing Heterogeneous Information Sources in TSIMMIS. In *In Proceedings of the AAAI Symposium on Information Gathering* (1995), pp. 61–64.

[70] Garcia-Molina, H., Papakonstantinou, Y., Quass, D., Rajaraman, A., Sagiv, Y., Ullman, J., Vassalos, V., and Widom, J. The TSIMMIS Approach to Mediation: Data Models and Languages. *Journal of Intelligent Information Systems 8*, 2 (1997), 117–132.

[71] Genesereth, M., and Fikes, R. Knowledge Interchange Format, Version 3.0 Reference Manual. Tech. rep., Computer Science Department, Stanford University, 1992.

[72] Ghawi, R., and Cullot, N. Database-to-Ontology Mapping Generation for Semantic Interoperability. In *Third International Workshop on Database Interoperability (InterDB 2007), held in conjunction with VLDB 2007* (Vienna, Austria, September 2007).

[73] Ghawi, R., and Cullot, N. Building Ontologies from Multiple Information Sources. In *15th Conference on Information and Software Technologies (IT2009)* (Kaunas, Lithuania, April 2009).

[74] Ghawi, R., and Cullot, N. Building Ontologies from XML Data Sources. In *1st International Workshop on Modelling and Visualization of XML and Semantic Web Data (MoViX '09), held in conjunction with DEXA'09* (Linz, Austria, September 2009).

[75] Ghawi, R., Poulain, T., Gomez-Carpio, G. V., and Cullot, N. OWSCIS: Ontology and Web Service Based Cooperation of Information Sources. In *Proceedings of the Third International IEEE Conference on Signal-Image Technologies and Internet-Based Systems (SITIS 2007)* (Shanghai, China, December 2007), IEEE Computer Society, pp. 246–253.

[76] Goh, C. H. *Representing and Reasoning about Semantic Conflicts in Heterogeneous Information Sources.* PhD thesis, Massachusetts Institute of Technology, 1997.

[77] Goh, C. H., Bressan, S., Madnick, S., and Siegel, M. Context Interchange: New Features and Formalisms for the Intelligent Integration of Information. *ACM Trans. Inf. Syst. 17*, 3 (1999), 270–293.

[78] Goh, C. H., Madnick, S. E., and Siegel, M. Context Interchange: Overcoming the Challenges of Large-Scale Interoperable Database Systems in a Dynamic Environment. In *Proceedings of the Third International Conference on Information and Knowledge Management (CIKM'94), Gaithersburg, Maryland* (1994), ACM, pp. 337–346.

[79] Gomez-Carpio, G. V., Abrouk, L., and Cullot, N. A Query Expansion Methodology in a Cooperation of Information Systems based on Ontologies. In *WEBIST 2009 - Proceedings of the Fifth International Conference on Web Information Systems and Technologies* (March 2009), INSTICC Press, pp. 256–262.

[80] Goni, A., Mena, E., and Illarramendi, A. Querying Heterogeneous and Distributed Data Repositories using Ontologies. In *Proceedings of the 7th European-Japanese Conference on Information Modelling and Knowledge Bases (IMKB'97)* (1997).

[81] Gray, P. M. D., Preece, A., Fiddian, N. J., Gray, W. A., Bench-Capon, T. J. M., Shave, M. J. R., Azarmi, N., Wiegand, M., Ashwell, M., Beer, M., Cui, Z., Diaz, B., Embury, S., Hui, K., Jones, A. C., Jones, D. M., Kemp, G. J. L., Lawson, E. W., Lunn, K., Marti, P., Shao, J., and Visser, P. R. S. KRAFT: Knowledge Fusion from Distributed Databases and Knowledge Bases. In *DEXA '97: Proceedings of the 8th International Workshop on Database and Expert Systems Applications* (Washington, DC, USA, 1997), IEEE Computer Society, p. 682.

[82] Gruber, T. R. Ontolingua: A Mechanism to Support Portable Ontologies. Tech. Rep. KSL 91-66, Knowledge Systems Laboratory, Stanford University, California, 1992.

[83] Gruber, T. R. A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition 5*, 2 (1993), 199–220.

[84] Gruber, T. R. Toward Principles for the Design of Ontologies Used for Knowledge Sharing. *International Journal of Human-Computer Studies 43*, 5-6 (1995), 907–928.

[85] Guarino, N., and Giaretta, P. Ontologies and Knowledge Bases: Towards a Terminological Clarification. In *Towards Very Large Knowledge Bases: Knowledge Building and Knowledge Sharing* (1995), N. J. I. Mars, Ed., IOS Press, Amsterdam, pp. 25–32.

[86] Haas, L. M., Hernandez, M. A., Ho, H., Popa, L., and Roth, M. Clio Grows Up: From Research Prototype to Industrial Tool. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data (SIGMOD'05)* (New York, NY, USA, 2005), ACM, pp. 805–810.

[87] Hakimpour, F., and Geppert, A. Ontologies: an Approach to Resolve Semantic Heterogeneity in Databases. Tech. rep., Department of Computer Science, University of Zurich, 2001.

[88] Harris, S., and Gibbins, N. 3store: Efficient bulk rdf storage. In *Proceedings of the 1st International Workshop on Practical and Scalable Semantic Systems (PSSS 2003)* (2003), R. Volz, S. Decker, and I. F. Cruz, Eds., vol. 89 of *CEUR Workshop Proceedings*, CEUR-WS.org, pp. 1–15.

[89] Heflin, J., Hendler, J. A., and Luke, S. SHOE: A Blueprint for the Semantic Web. In *Spinning the Semantic Web: Bringing the World Wide Web to Its Full Potential* (2003), D. Fensel, J. A. Hendler, H. Lieberman, and W. Wahlster, Eds., MIT Press, pp. 29–63.

[90] Hewlett-Packard Development Company. Jena - A Semantic Web Framework for Java. available: http://jena.sourceforge.net/index.html, 2006.

[91] Horrocks, I., Li, L., Turi, D., and Bechhofer, S. The Instance Store: Description Logic Reasoning with Large Numbers of Individuals. In *in 17 International Workshop on Description Logics (DL 2004)* (2004), pp. 31–40.

[92] ISO 8879:1986. Information Processing – Text and Office Systems – Standard Generalized Markup Language (SGML). Standard No. ISO 8879:1986, International Organization for Standardization, Geneva, Switzerland, 1986.

[93] Jeusfeld, M. A., and Johnen, U. A. An Executable MetaModel for Re-Engineering of Database Schemas. In *Proceedings in the 13th International Conference of Entity-Relationship Approach, Manchester, UK* (1994).

[94] Karp, P. D., Chaudhri, V., and Thomere, J. XOL: An XML-Based Ontology Exchange Language. Tech. rep., Knowledge Systems Laboratory, Stanford University, California, 1999.

[95] Karp, P. D., Myers, K. L., and Gruber, T. R. The Generic Frame Protocol. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95, Montreal, Quebec, Canada* (August 1995), C. Mellish, Ed., Morgan Kaufmann Publishers, San Francisco, California, pp. 768–774.

[96] Karvounarakis, G., Alexaki, S., Christophides, V., Plexousakis, D., and Scholl, M. RQL: A Declarative Query Language for RDF. In *Proceedings of the Eleventh International World Wide Web Conference* (2002).

[97] Katki, F., McMonegal, L., Meyer, B., Lane, J., Wilson, P., Radatz, J., Yee, M., Porteous, H., and Springsteel, F., Eds. *IEEE Standard Computer Dictionary: Compilation of IEEE Standard Computer Glossaries*. Institute of Electrical and Electronics Engineers Inc., 1991.

[98] Keim, D., Kriegel, H., and Miethsam, A. Query Translation Supporting the Migration of Legacy Database into Cooperative Information Systems. In *Proceedings of the 2nd International Conference on Cooperative Information Systems* (1994), pp. 203–214.

[99] Kifer, M., Lausen, G., and Wu, J. Logical Foundations of Object-Oriented and Frame-Based Languages. *Journal of the ACM 42*, 4 (1995), 741–843.

[100] Kim, W., Choi, I., Gala, S., and Scheevel, M. On Resolving Schematic Heterogeneity in Multidatabase Systems. *Distributed and Parallel Databases 1*, 3 (1993), 251–279.

[101] Kim, W., and Seo, J. Classifying Schematic and Data Heterogeneity in Multi Database Systems. *IEEE Computer 24* (Dec. 1991), 12–18.

[102] Kirk, T., Levy, A. Y., Sagiv, Y., and Srivastava, D. The Information Manifold. In *Proceedings of the AAAI 1995 Spring Symp. on Information Gathering from Heterogeneous, Distributed Enviroments* (1995), pp. 85–91.

[103] Klein, M., Fensel, D., van Harmelen, F., and Horrocks, I. The Relation between Ontologies and Schema-Languages: Translating OIL-Specifications to XML-Schema. In *Proceedings of the ECAI'00 workshop on applications of ontologies and problem-solving methods* (Berlin, August 2000).

[104] Klyne, G., and Carroll, J. J. Resource Description Framework (RDF): Concepts and Abstract Syntax. Tech. rep., W3C, February 2004. W3C Recommendation.

[105] Kobeissy, N., Genet, M. G., and Zeghlache, D. Mapping XML to OWL for Seamless Information Retrieval in Context-Aware Environments. In *International Conference on Pervasive Services* (Los Alamitos, CA, USA, 2007), IEEE Computer Society, pp. 361–366.

[106] Konstantinou, N., Spanos, D.-E., Chalas, M., Solidakis, E., and Mitrou, N. VisAVis: An Approach to an Intermediate Layer between Ontologies and Relational Database Contents. In *Proceedings of the CAISE*06 Third International Workshop on Web Information Systems Modeling (WISM '06)* (June 2006), vol. 239 of *CEUR Workshop Proceedings*, CEUR-WS.org.

[107] Kunfermann, P., and Drumm, C. Lifting XML Schemas to Ontologies - The Concept Finder Algorithm. In *MEDIATE2005* (2005), M. Hepp, A. Polleres, F. van Harmelen, and M. R. Genesereth, Eds., vol. 168 of *CEUR Workshop Proceedings*, CEUR-WS.org, pp. 113–122. online http://CEUR-WS.org/Vol-168/MEDIATE2005-paper8.pdf.

[108] Lakshmanan, L. V. S., Sadri, F., and Subramian, I. N. On The Logical Foundations Of Schema Integration And Evolution In Heterogeneous Database Systems. In *Deductive and Object-Oriented Database, 3rd International Conference, LNCS 760, Phoenix, Arizona, USA* (1993).

[109] Lassila, O., and McGuinness, D. The Role of Frame-Based Representation on the Semantic Web. Tech. Rep. KSL-01-02, Knowledge Systems Laboratory, Stanford University, California, USA, Stanford, 2001.

[110] Lenat, D. B., and Guha, R. V. *Building Large Knowledge-Based Systems; Representation and Inference in the Cyc Project.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[111] Lenzerini, M. Data Integration: A Theoretical Perspective. In *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems* (New York, NY, USA, 2002), ACM, pp. 233–246.

[112] Ludascher, B., Himmeroder, R., Lausen, G., May, W., and Schlepphorst, C. Managing Semistructured Data with FLORID: A Deductive Object-Oriented Perspective. *Information Systems 23*, 8 (1998), 589–613.

[113] Luke, S., and Heflin, J. SHOE 1.01. Proposed Specification. Tech. rep., Parallel Understanding Systems Group, Department of Computer Science, University of Maryland., February 2000.

[114] MacGregor, R. M. A Deductive Pattern Matcher. In *Proceedings of AAAI-88, St. Paul, Minnesota.* (1988), pp. 403–408.

[115] MacGregor, R. M. Inside the LOOM Classifier. In *SIGART Bulletin* (1991), vol. 2(3), pp. 70–76.

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

[117] Maler, E. Schema Design Rules for UBL... and Maybe for You. In *IDEALLINACE XML 2002 Conference and Exposition* (2002).

[118] McGuinness, D. L., and van Harmelen, F. OWL Web Ontology Language Overview. W3C recommendation, W3C, February 2004.

[119] Mena, E., Kashyap, V., Illarramendi, A., and Sheth, A. Managing Multiple Information Sources through Ontologies: Relationship between Vocabulary Heterogeneity and Loss of Information. In *In Proceedings of Knowledge Representation Meets Databases (KRDB'96), ECAI'96 conference* (1996), pp. 50–52.

[120] Mena, E., Sheth, A., and Illarramendi, A. OBSERVER: An Approach for Query Processing in Global Information Systems based on Interoperation across Pre-existing Ontologies. In *Cooperative Information Systems* (1996), IEEE Computer Society Press, pp. 14–25.

[121] Miller, R. J. Using Schematically Heterogeneous Structures. *SIGMOD Rec. 27*, 2 (1998), 189–200.

[122] Motta, E. *Reusable Components for Knowledge Modelling: Case Studies in Parametric Design Problem Solving*, vol. 53 of *Frontiers in Artificial Intelligence and Applications.* IOS Press, Amsterdam, The Netherlands, 1999.

[123] Neches, R., Fikes, R., Finin, T., Gruber, T., Senator, T., and Swartout, W. Enabling Technologies for Knowledge Sharing. *AI Magazine 12(3)* (1991), 36–56.

[124] Nicolle, C., Simon, J.-C., and Yetongnon, K. Interoperability of Information Systems. In *Encyclopedia of Information Science and Technology (III)*, M. Khosrow-Pour, Ed., vol. I-V. IDEA Group Publishing, 2005, pp. 1651–1650.

[125] Noy, N. F., Fergerson, R. W., and Musen, M. A. The Knowledge Model of Protégé-2000: Combining Interoperability and Flexibility. In *EKAW '00: Proceedings of the 12th European Workshop on Knowledge Acquisition, Modeling and Management* (London, UK, 2000), Springer-Verlag, pp. 17–32.

[126] Noy, N. F., and Klein, M. Ontology Evolution: Not the Same as Schema Evolution. *Knowledge and Information Systems 6*, 4 (July 2004), 428–440.

[127] Noy, N. F., and Musen, M. A. PROMPT: Algorithm and Tool for Automated Ontology Merging and Alignment. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence* (2000), AAAI Press / The MIT Press, pp. 450–455.

[128] OMG. The Common Object Request Broker: Architecture and Specification. Tech. Rep. 91.12.1 rev 1.1, Object Management Group, 1992.

[129] Ozsu, M. T., and Valduriez, P. *Principles of Distributed Database Systems*, 2nd ed. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1999.

[130] Pan, Z., and Heflin, J. DLDB: Extending Relational Databases to Support Semantic Web Queries. In *Proceedings of Workshop on Practical and Scaleable Semantic Web Systems (PSSS 2003)* (2003).

[131] Poulain, T. *Une Approche Orientée Sémantique pour l'Interrogation d'une Coopération de Systèmes d'Information basée sur des Ontologies.* PhD thesis, Université de Bourgogne, Dijon, France, 2009.

[132] Poulain, T., and Cullot, N. Ontology Mappings for Information Systems Cooperation. In *Journées Francophones sur les Ontologies (JFO2008)* (Lyon, France, December 2008), D. Benslimane, C. Roche, and S. Spaccapietra, Eds., ACM, pp. 1–12.

[133] Poulain, T., Cullot, N., and Yetongnon, K. Ontology Mapping Specification in Description Logics for Cooperative Systems. In *Proceedings of the 1st International Conference on Signal-Image Technology and Internet-Based Systems (SITIS 2005)* (Yaounde, Cameroon, 2005), Dicolor Press, pp. 240–245.

[134] Poulain, T., Cullot, N., and Yetongnon, K. Similarity Estimation Module for OWSCIS. In *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC 2008)* (Fortaleza, Ceara, Brazil, March 2008), ACM, pp. 2305–2309.

[135] Powell, G. *Beginning XML Databases.* Wrox Press Ltd., Birmingham, UK, 2006.

[136] Preece, A., Hui, K., and Gray, P. KRAFT: Supporting Virtual Organisations through Knowledge Fusion. In *Artificial Intelligence for Electronic Commerce: papers from the AAAI-99 Workshop* (1999), T. Finin and B. Grosof, Eds., AAAI Press, pp. 33–38.

[137] Preece, A. D., Hui, K., Gray, A., Marti, P., Bench-Capon, T., Jones, D., and Cui, Z. The KRAFT Architecture for Knowledge Fusion and Transformation. *Knowledge Based Systems 13*, 2-3 (2000), 113–120.

[138] Prud'Hommeaux, E., and Seaborne, A. SPARQL Query Language for RDF. Tech. rep., World Wide Web Consortium, January 2008.

[139] Reese, G. *Database Programming with JDBC and Java, Second Edition.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2000.

[140] Rodrigues, T., Rosa, P., and Cardoso, J. Mapping XML to Existing OWL Ontologies. In *International Conference WWW/Internet 2006* (2006), P. Isaas, M. B. Nunes, and I. J. Martnez, Eds., pp. 72–77.

[141] Seaborne, A. RDQL - A Query Language for RDF. Tech. rep., W3C, 2004.

[142] Shadbolt, N., Motta, E., and Rouge, A. Constructing Knowledge-Based Systems. *IEEE Software 10*, 6 (1993), 34–38.

[143] Sheth, A. *Changing Focus on Interoperability in Information Systems: from System, Syntax, Structure to Semantic.* Kluwer Academic Publishers, 1999, pp. 5–29. In M.F. Goodchild, M.J. Egenhofer, R. Fegeas, and C.A. Kotman, editors. *Interoperating Geographic Information Systems.* Kluwer Academic Publishers, Norwell, MA, pages 5–29, 1999.

[144] Sheth, A. P., and Larson, J. A. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys 22* (1990), 183–236.

[145] Siegel, M., and Madnick, S. E. A Metadata Approach to Resolving Semantic Conflicts. In *VLDB '91: Proceedings of the 17th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1991), Morgan Kaufmann Publishers Inc., pp. 133–145.

[146] Simon, J.-c., and Nicolle, C. Building Web-Services from RDBMS. In *2nd IEEE International Symposium on Signal Processing and Information Technology, Marrakesh, Morocco* (Marrakesh, Morocco, December 2002), pp. 122–126.

[147] Sowa, J. F. *Knowledge Representation: Logical, Philosophical, and Computational Foundations.* Course Technology, August 1999.

[148] Stojanovic, L., Stojanovic, N., and Volz, R. Migrating Data-Intensive Web Sites into the Semantic Web. In *Proceedings of the 2002 ACM symposium on Applied computing (SAC'02:)* (New York, NY, USA, 2002), ACM, pp. 1100–1107.

[149] Stojanovic, N., Stojanovic, L., and Volz, R. A Reverse Engineering Approach for Migrating Data-intensive Web Sites to the Semantic Web. In *Proceedings of the IFIP 17th World Computer Congress - TC12 Stream on Intelligent Information Processing* (Deventer, The Netherlands, 2002), Kluwer, B.V., pp. 141–154.

[150] Stuckenschmidt, H., and Harmelen, F. v. *Information Sharing on the Semantic Web.* SpringerVerlag, 2004.

[151] Stuckenschmidt, H., Schlieder, C., Visser, U., Vogele, T., and Neumann, H. Spatial Reasoning for Information Brokering. In *Proceedings of the Fourteenth International Florida Artificial Intelligence Research Society Conference* (2001), AAAI Press, pp. 568–573.

[152] Stuckenschmidt, H., and Wache, H. Context Modelling and Transformation for Semantic Interoperability. In *Knowledge Representation Meets Databases (KRDB 2000) - CEUR Workshop Proceedings* (2000), M. Bouzeghoub, M. Klusch, W. Nutt, and U. Sattler, Eds.

[153] Studer, R., Benjamins, R., and Fensel, D. Knowledge Engineering: Principles and Methods. *Data and knowledge engineering 25* (1998), 161–197.

[154] Sun, W., and Liu, D.-X. Using Ontologies for Semantic Query Optimization of XML Database. In *Knowledge Discovery from XML Documents, First International Workshop, KDXD 2006* (2006), vol. 3915 of *Lecture Notes in Computer Science*, Springer, pp. 64–73.

[155] Thompson, H. S., Beech, D., Maloney, M., and Mendelsohn, N. XML Schema Part 1: Structures Second Edition. Tech. rep., W3C, October 2004. W3C Recommendation.

[156] Tsichritzis, D., and Klug, A. C. The ANSI/X3/SPARC DBMS Framework Report of the Study Group on Dabatase Management Systems. *Information Systems 3*, 3 (1978), 173–191.

[157] UDDI. UDDI Version 3.0.2, UDDI Spec Technical Committee Draft. Tech. rep., OASIS, October 2004.

[158] Uschold, M., and Gruninger, M. Ontologies: Principles, Methods and Applications. *Knowledge Engineering Review 11* (1996), 93–136.

[159] Visser, U., Stuckenschmidt, H., Wache, H., and Vgele, T. Enabling Technologies for Interoperability. In *14th International Symposium of Computer Science for Environmental Protection, Bonn, Germany* (2000), U. Visser and H. Pundt, Eds., TZI, University of Bremen, pp. 35–46.

[160] Volz, R., Handschuh, S., Staab, S., and Studer, R. OntoLiFT Demonstrator. Tech. Rep. D12, WonderWeb project deliverable, 2004.

[161] W3C. Web Services Architecture Requirements, February 2004.

[162] W3C. SOAP Version 1.2 Part 0: Primer (Second Edition), April 2007. W3C Recommendation.

[163] W3C. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. W3c recommendation, W3C, March 2007.

[164] Wache, H., and Stuckenschmidt, H. Practical Context Transformation for Information System Interoperability. In *CONTEXT '01: Proceedings of the Third International and Interdisciplinary Conference on Modeling and Using Context* (London, UK, 2001), Springer-Verlag, pp. 367–380.

[165] Wache, H., Vogele, T., Visser, U., Stuckenschmidt, H., Schuster, G., Neumann, H., and Hubner, S. Ontology-based Integration of Information - a Survey of Existing Approaches. In *IJCAI–01 Workshop: Ontologies and Information Sharing* (2001), pp. 108–117.

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

[167] Woelk, D., and Tomlinson, C. The InfoSleuth Project: Intelligent Search Management via Semantic Agents. Tech. rep., Microeclectronics and Computer Technology Corporation (MCC), September 1994. Also available in the Electronic Proceedings of the Second World Wide Web Conference '94.

[168] Xiao, H., and Cruz, I. F. Integrating and Exchanging XML Data Using Ontologies. *Journal on Data Semantics VI: Special Issue on Emergent Semantics 4090* (2006), 67–89.

[169] Xu, L., and Embley, D. W. Combining the Best of Global-as-View and Local-as-View for Data Integration. In *Information Systems Technologies and Its Applications - ISTA* (2003), pp. 123–136.

[170] Yan, L.-L., and Ling, T. W. Translating Relational Schema With Constraints Into OODB Schema. In *Proceedings of the IFIP WG 2.6 Database Semantics Conference on Interoperable Database Systems (DS-5)* (Amsterdam, The Netherlands, The Netherlands, 1993), North-Holland Publishing Co., pp. 69–85.

[171] Yang, K., Steele, R., and Lo, A. An Ontology for XML Schema to Ontology Mapping Representation. In *Proceedings of the 9th International Conference on Information Integration and Web-based Application & Services, iiWAS* (2007), G. Kotsis, D. Taniar, E. Pardede, and I. K. Ibrahim, Eds., vol. 229 of *books@ocg.at*, Austrian Computer Society, pp. 101–111.