Master Degree in Computer Science
Mestrado em Engenharia Informática

# A SPARQL Query Engine over Web Ontologies using Contextual Logic Programming

Nuno Alexandre de Jesus Lopes
<nl@uevora.pt>

**Supervisor:** Salvador Pinto Abreu

*This thesis does not include appreciation nor suggestions made by the jury.*
*Esta dissertação não inclui as críticas e sugestões feitas pelo júri.*

Évora, October 2007

# Abstract


In the World Wide Web a machine can easily process the structure of resources. However, more advanced techniques are necessary to determine the semantic contents of such resources.

The Semantic Web is an enhancement of the Web which aims to overcome this difficulty. Annotating Web resources with information about their contents, described using a formal language, helps machines process the semantics of the resource.

Integrating that semantic information, which can be described using OWL, with a Contextual Logic Programming framework provides a clear and simple way to represent and query an ontology. The main contributions of this work to that purpose are:

- A system capable of representing OWL DL ontologies and performing queries over that representation

- A SPARQL query answering module: this makes the system available to a wide range of users and automated processes, enabling the possibility of advertising it as a web service.

# Resumo


SPARQL para Interrogação de Ontologias Web com Programação em Lógica Contextual

Na World Wide Web os programas informáticos conseguem processar fácilmente a estrutura das páginas. Contudo, são necessárias técnicas avançadas para determinar o conteúdo semântico dessas páginas.

A Semantic Web é uma extensão da Web que tenta ultrapassar esta dificuldade. Criar anotações nas páginas com informação acerca do seu conteúdo, descrita numa linguagem formal, ajuda o processamento automatizado da semântica da página.

A integração dessa informação semântica, que pode ser descrita através de OWL, com uma framework de Programação em Lógica Contextual disponibiliza uma maneira clara e simples de representar e interrogar ontologias. As principais contribuições deste trabalho para esse objectivo são:

- Um sistema capaz de representar ontologias OWL DL e de realizar interrogações baseadas nessa representação
- Um módulo de resposta a interrogações SPARQL: permite disponibilizar o sistema a um elevado número de utilizadores e processos automáticos e possibilita anunciá-lo como um serviço Web.

# Acknowledgments

I would like to thank Cláudio Fernandes for all the efforts in the development of our work and Salvador Abreu for all the guidance provided. Thank you also to all the people that helped me by reviewing this thesis: Luís Simões, Nuno Morgadinho, José Saias and Tiago Sousa.

Last but not least, to my family and friends. Thank you for all the strength and support provided and for always believing in me.

# Acronyms and Definitions

**Arity** = The number of arguments of a predicate or unit

**Context** = A ordered sequence of units where a CxLP goal is executed

**CxLP** = Contextual Logic Programming

**Functor** = The name of a predicate

**HTML** = HyperText Markup Language

**ISCO** = Information System COnstruction, a framework for accessing relational databases from Prolog

**OWL** = Web Ontology Language

**Ontology** = A formal description about concepts of a specific domain. Can be described using OWL

**PiLLoW** = A library for handling WWW documents (HTML, XML) from Prolog

**Predicate** = A logical assertion, possibly including conditions for validity

**RDF** = Resource Description Framework

**RDFS** = RDF Schema

**SPARQL** = SPARQL Protocol and RDF Query Language

**Unit** = The CxLP representation of a set of Prolog predicates

**W3C** = World Wide Web Consortium

**XML** = eXtensible Markup Language

# Contents

**1 Introduction** **1**
1.1 The Semantic Web . . . . . 1
1.2 Motivation and Objectives . . . . . 2
1.3 Related Work . . . . . 2
1.4 Guide . . . . . 5

**2 Semantic Web** **6**
2.1 What is the Semantic Web? . . . . . 6
2.2 Foundations of the Semantic Web . . . . . 7
2.2.1 A Layered Approach . . . . . 8
2.2.2 Metadata . . . . . 10
2.2.3 Ontologies . . . . . 11
2.3 Web Languages . . . . . 12
2.4 Query Languages . . . . . 16
2.5 Applications for the Semantic Web . . . . . 18
2.5.1 Improving Web search . . . . . 18
2.5.2 Web as a database . . . . . 18
2.5.3 Ubiquitous Computing and Web services . . . . . 19
2.6 Conclusion . . . . . 19

**3 Contextual Logic Programming** **20**
3.1 Logic Programming . . . . . 20
3.1.1 Logic Programming and the Semantic Web . . . . . 21
3.2 Contextual Logic Programming . . . . . 22
3.2.1 GNU Prolog/CX . . . . . 22
3.3 ISCO . . . . . 24
3.4 PiLLoW . . . . . 25
3.5 Conclusion . . . . . 26

**4 The XPTO system** **27**
4.1 Parsing an ontology . . . . . . . . . . . . . . . . . . . . . . . . . 28
4.2 Representation of an ontology . . . . . . . . . . . . . . . . . . . 29
4.2.1 Ontology representation . . . . . . . . . . . . . . . . . . 30
4.2.2 Name analysis . . . . . . . . . . . . . . . . . . . . . . . . 35
4.2.3 Unit generation and loading . . . . . . . . . . . . . . . . 38
4.3 Querying an ontology . . . . . . . . . . . . . . . . . . . . . . . . 39
4.3.1 Units for refining ontology queries . . . . . . . . . . . . 40
4.3.2 Native Prolog query representation . . . . . . . . . . . 42
4.4 Example Use Cases . . . . . . . . . . . . . . . . . . . . . . . . . 43
4.4.1 SPARQL Query examples . . . . . . . . . . . . . . . . . 43
4.4.2 Database integration using ISCO . . . . . . . . . . . . 45
4.5 Benchmarks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
4.5.1 XML Parsers . . . . . . . . . . . . . . . . . . . . . . . . 48
4.5.2 Ontology representation . . . . . . . . . . . . . . . . . . 53
4.5.3 XPTO time analysis . . . . . . . . . . . . . . . . . . . . 54
4.6 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55

**5 SPARQL Query Engine** **56**
5.1 Querying in SPARQL . . . . . . . . . . . . . . . . . . . . . . . . 56
5.1.1 SPARQL query elements . . . . . . . . . . . . . . . . . . 57
5.1.2 Query forms and results . . . . . . . . . . . . . . . . . . 60
5.1.3 Solution modifiers . . . . . . . . . . . . . . . . . . . . . . 61
5.1.4 Querying OWL ontologies using SPARQL . . . . . . . 62
5.2 Representation of a SPARQL query . . . . . . . . . . . . . . . . 62
5.2.1 Element representation . . . . . . . . . . . . . . . . . . . 62
5.2.2 Query representation . . . . . . . . . . . . . . . . . . . . 64
5.2.3 SPARQL parser . . . . . . . . . . . . . . . . . . . . . . . 66
5.3 SPARQL resolution system . . . . . . . . . . . . . . . . . . . . . 67
5.3.1 Unit description . . . . . . . . . . . . . . . . . . . . . . . 68
5.3.2 Unimplemented features . . . . . . . . . . . . . . . . . . 72
5.4 Returning Query Results . . . . . . . . . . . . . . . . . . . . . . 73
5.4.1 Select query . . . . . . . . . . . . . . . . . . . . . . . . . 73
5.4.2 Ask query . . . . . . . . . . . . . . . . . . . . . . . . . . 73
5.5 Examples . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
5.5.1 Using SPARQL to query a relational database . . . . . 75
5.5.2 SPARQL Web service . . . . . . . . . . . . . . . . . . . . 76
5.6 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77

**6 Conclusion** **78**

# List of Figures


2.1 A layered approach to the Semantic Web . . . . . . . . . . . . 9
2.2 RDF XML syntax example . . . . . . . . . . . . . . . . . . . . . . 13
2.3 RDF Turtle syntax example . . . . . . . . . . . . . . . . . . . . . 14

3.1 Example unit: `teacher/0` . . . . . . . . . . . . . . . . . . . . . 22
3.2 Example unit: `teacher/3` . . . . . . . . . . . . . . . . . . . . . 23
3.3 Example class definitions in ISCO . . . . . . . . . . . . . . . . . 25
3.4 PiLLoW term example . . . . . . . . . . . . . . . . . . . . . . . . 25
3.5 PiLLoW output . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26

4.1 XML example: OWL Class definition . . . . . . . . . . . . . . . 29
4.2 Prolog XML Representation of the example in Figure 4.1 . . . 30
4.3 Ontology representation schema: units . . . . . . . . . . . . . . 31
4.4 OWL property definition . . . . . . . . . . . . . . . . . . . . . . . 32
4.5 Prolog representation of Figure 4.4 . . . . . . . . . . . . . . . . 32
4.6 `IceWine` class unit (first version, partial) . . . . . . . . . . . . 35
4.7 Class `item` goals example . . . . . . . . . . . . . . . . . . . . . 35
4.8 `AllValuesFrom` example . . . . . . . . . . . . . . . . . . . . . . 36
4.9 `AllDifferent` example . . . . . . . . . . . . . . . . . . . . . . . . 37
4.10 `AllDifferent` representation . . . . . . . . . . . . . . . . . . . . 37
4.11 ontology query (direct access) . . . . . . . . . . . . . . . . . . . 40
4.12 ontology query (`property/2` example) . . . . . . . . . . . . . . 40
4.13 Example of a complete query context . . . . . . . . . . . . . . . 40
4.14 unit `access/0` (partial) . . . . . . . . . . . . . . . . . . . . . . . 41
4.15 predicate definition example . . . . . . . . . . . . . . . . . . . . 43
4.16 generated predicate . . . . . . . . . . . . . . . . . . . . . . . . . . 43
4.17 SPARQL example: Query 1 . . . . . . . . . . . . . . . . . . . . . 44
4.18 Query 1 - XPTO syntax . . . . . . . . . . . . . . . . . . . . . . . 44
4.19 SPARQL example: Query 4 . . . . . . . . . . . . . . . . . . . . . 45
4.20 Query 4 - XPTO syntax . . . . . . . . . . . . . . . . . . . . . . . 45
4.21 SPARQL example: Query 9 . . . . . . . . . . . . . . . . . . . . . 45
4.22 Query 9 - XPTO syntax . . . . . . . . . . . . . . . . . . . . . . . 46

4.23 Periodic table ontology example: group_10 . . . . . . . . . . . 46
4.24 Database table example: element . . . . . . . . . . . . . . . . . 47
4.25 Query example (ontologies and ISCO) . . . . . . . . . . . . . 47
4.26 Expat Library vs Libxml2 . . . . . . . . . . . . . . . . . . . . 50
4.27 Speedup graph . . . . . . . . . . . . . . . . . . . . . . . . . . 53

5.1 SPARQL query engine architecture . . . . . . . . . . . . . . . . 57
5.2 SPARQL query example . . . . . . . . . . . . . . . . . . . . . . 61
5.3 Query example (simple select) . . . . . . . . . . . . . . . . . . 64
5.4 Generated context (partial) for the query in Figure 5.3 . . . . 64
5.5 Query example . . . . . . . . . . . . . . . . . . . . . . . . . . 66
5.6 Context generated for the query in Figure 5.5 . . . . . . . . . 67
5.7 Auxiliary Parser structures . . . . . . . . . . . . . . . . . . . . 68
5.8 Unit triple/3 . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
5.9 limit and offset query example . . . . . . . . . . . . . . . . . 71
5.10 XML output of the query example in Figure 5.3 . . . . . . . . 74
5.11 XML output for an ASK query . . . . . . . . . . . . . . . . . . 74
5.12 ISCO definition of the relation student . . . . . . . . . . . . . 75
5.13 using SPARQL to query a relational database . . . . . . . . . 76
5.14 Generated context for the query in Figure 5.13 . . . . . . . . . 76
5.15 SQL queries generated for the context in Figure 5.14 . . . . . 77

# List of Tables

4.1 Libxml2 and Libexpat comparison (seconds) . . . . . . . . . . 49
4.2 Benchmark results (seconds) . . . . . . . . . . . . . . . . . . . . 51
4.3 Speedups results. Jena used as reference. . . . . . . . . . . . . 52
4.4 Speedups results (percentage). Jena used as reference. . . . . . 52
4.5 Time (in seconds) of representing the ontologies . . . . . . . . 54
4.6 Performance gain of representing the ontologies . . . . . . . . 54
4.7 Average time of each part of the representation time . . . . . 55

5.1 SPARQL Query Language Structure (adapted) . . . . . . . . . 58
5.2 Query context structure . . . . . . . . . . . . . . . . . . . . . . 65

# Chapter 1

# Introduction

The system being presented here, XPTO[1], enables accessing OWL (Web Ontology Language) ontologies from within a Contextual Logic Programming environment, namely GNU Prolog/CX. It also allows to integrate these ontologies in the running program enabling using them as a part of the computation.

Also presented is a component of the system that enables it to answer queries formulated using the SPARQL query language and thus presenting the possibility of making the system visible to the World Wide Web though a Web Service.

Throughout this work, Logic Programming (more specifically Contextual Logic Programming) is used to represent and query the ontologies in the XPTO system core and also to represent and evaluate the queries in the SPARQL query answering module.

## 1.1 The Semantic Web

The main purpose of the Semantic Web [BLHL01] is to provide *machine understandable* Web resources, allowing the information they contain to be accessed and processed by machines alongside human users.

In order to achieve this, it will be necessary for Web Resources to be annotated with information describing their contents, using a standard description language that can be understood by humans and easily processed by machines.

The commonly used annotation language is RDF and, based on RDF, the OWL ontology language was developed. An ontology is a description of definitions and concepts about a certain domain. OWL is a language

[1]XPTO is a recursive acronym that stands for *XPTO Prolog Translation of Ontologies*.

compatible with current Web standards (XML, RDF and RDFS) and whose semantics are formally defined [DSB+04].

## 1.2 Motivation and Objectives

The purpose of this work is to use Logic Programming, more specifically Contextual Logic Programming, as a mediator framework for Semantic Web agents, in which knowledge representation for ontology documents and other sources of information can be integrated in a transparent manner. For example using the ISCO framework [AN06] it is possible to integrate relational databases and Web ontologies.

Contextual Logic Programming is an extension to Logic Programming that intends to introduce modular programming. The adopted framework, GNU Prolog/CX, described in [AN06] makes use of persistence and program structuring through the use of contexts [AD03].

The main objective of the XPTO system is to be able to represent and query web ontologies from the perspective of Contextual Logic Programming.

After transforming the information of an ontology into GNU Prolog/CX units and predicates, it is possible to use these definitions in a Prolog computation and access the ontology. This framework can be used to provide a front end that can act as a SPARQL web agent. This front end can receive a SPARQL query about a known ontology, process it against the internal representation and return the corresponding results.

Parts of this work have previously been presented: an initial description of the system was shown in [FLA07]. The examples and use cases for the system described in Section 5.5 were presented in [LFA07].

## 1.3 Related Work

Other available systems provide similar capabilities to XPTO. Either in the representation of the ontologies, SPARQL query engines or in both aspects. Some of these systems are briefly introduced next:

### Thea

Thea [Van06] is an OWL parser implemented in Prolog. It uses The SWI-Prolog Semantic Web library to parse the OWL ontologies into RDF triples

and then builds the representation based on these results. The ontology is represented as Prolog terms and its structure is further described in [Van07].

Known issues and limitations are:

- Thea does not make any inferences nor does it check the consistency of the OWL ontology.
- It parses OWL Full, DL and Lite ontologies but it does not validate them.
- It does not support the `owl:import` directive: the imported ontology is not parsed automatically.

## Racer

Racer [Sof07, HM01] is an OWL reasoner and inference engine for the Semantic Web. It enables applications to query OWL ontologies implementing the candidate standard OWL querying language OWL-QL.

Implemented in Common Lisp, Racer is able to start multiple reasoners on the local machine and distribute its load among the Racer instances. It also uses query caching, each query sent by clients and each answer to that query obtained through reasoning can be cached by the system.

**Protégé** Protégé [Pro06] is a platform that provides tools to construct ontologies. Through its Protégé-OWL plugin, it enables users to build ontologies for the Semantic Web [KMR04] and allows integration with reasoners such as Racer.

## Jena

Jena [Jen06] is an open source Java framework for the Semantic Web developed at the *HP Labs Semantic Web Programme*. Jena includes a RDF API, an OWL API and a SPARQL query engine allowing Java programmers to access OWL ontologies in a simple manner.

The SPARQL query engine present in Jena, known as ARQ [2], allows Jena from within its framework, to query an external SPARQL agent and process the returning results.

[2] Documentation about ARQ can be found at `http://jena.sourceforge.net/ARQ/documentation.html`

**Ontology inference** Jena can use different reasoners to infer new triples. All inferred information is stored as new triples thus exposing them to the queries.[3]

**Persistence Storage** In addition to in-memory storage, Jena provides persistent storage of RDF documents in relational databases. The persistence subsystem supports an interface for SPARQL queries that dynamically generates SQL queries.

## Pellet

Pellet [SPG+07] is an open source reasoner for the OWL DL ontology language developed at the Mindswap Lab of the University of Maryland.

Pellet contains a query engine which supports answering queries formulated using SPARQL and supports reasoning with multiple ontologies.

Some of other important features include:

**Consistency checking:** No contradictory facts are present in the ontologies.

**Concept satisfiability:** Checks the possibility of the classes to have any instances.

**Classification:** Computes all the ontology hierarchy.

**Realization:** Computes the direct types of each individual

## F-OWL

F-OWL is implemented using Flora-2 which is a extension of F-logic, a logic based language with some aspects of Object-Oriented Programming. F-OWL is a rule based ontology inference engine for OWL. F-OWL makes use of mechanisms of the underlying technology (XSB Prolog) such as tabling for result caching.

F-OWL supports ontology inferences based on the OWL-Lite language. Only a few number of inference rules have been prototyped for experimenting OWL DL and OWL Full ontology inferences.

[3]Further information is available at `http://jena.sourceforge.net/inference/`

## 1.4 Guide

The remainder of the thesis is organised as follows:

**Chapter 2** presents the Semantic Web and some of the associated technologies

**Chapter 3** introduces Contextual Logic Programming, the used implementation GNU Prolog/CX and other used frameworks

**Chapter 4** describes the XPTO core system (that consists of the ontology mapping framework). This chapter corresponds to the description of the work developed in cooperation with Cláudio Fernandes

**Chapter 5** presents a component of XPTO, more specifically the implementation of a SPARQL query answering system

**Chapter 6** lays out the conclusions and indicates the work that still needs to be developed

# Chapter 2

# Semantic Web

This chapter briefly discusses a vision for the Semantic Web, a possible structure for it (Section 2.2) and introduces associated technologies in Sections 2.3 and 2.4. Finally, some possible applications for the Semantic Web are presented in Section 2.5.

## 2.1 What is the Semantic Web?

> "The Semantic Web is not a separate Web but an extension of the current one, in which information is given well-defined meaning, better enabling computers and people to work in cooperation".

This quote, taken from "The Semantic Web" [BLHL01] outlines the purpose and ideal of the Semantic Web: to allow machines, along with humans, to make better use of the information that can be found in the Web, infer meaning and knowledge from that information in order to assist human users in their daily activities.

This is not a new vision, already having been hinted in the first World Wide Web Conference (1994) by Tim Berners-Lee as stated in [SBLH06], and later in *Weaving the Web* [BLF99]. *"The Semantic Web"* [BLHL01] describes a possible scenario for the Semantic Web and the technologies that could be used to achieve it. The scenario is that of intelligent agents able to communicate with other agents and Web resources (such as web pages or web services) to accomplish useful goals for users.

Currently the web is designed for users to navigate the pages and extract from them the intended meaning. It is the user that navigates the pages,

uses a search engine, buys products, *etc.* The focus is mostly on the human user. The Semantic Web brings forth a shift of focus in order to also include the software programs. It would be more efficient if the user could perform a query that would then proceed on its own instead of manually browsing through pages to retrieve the information or achieve the intended action.

## 2.2 Foundations of the Semantic Web

For the Semantic Web to follow the current web model there are certain principles and key aspects of the World Wide Web that should be respected [KM02]. The main principles are described bellow:

**URI-style addressing** The Semantic Web can use identifiers to refer to things in the physical world, such as people and places. A URI (Uniform Resource Identifier) can be created to identify something in the physical world or things can be referred indirectly, e.g., it is possible to identify a person by referring to the e-mail address of that person.

**Annotated resources and links** The Web consists of resources and links. For the information in these resources to be more easily processed by machines there should be available, in the resources and links, information about its contents or the relation between the two resources (in the case of links).

**Partial information is tolerated** It is essential for the Semantic Web to be able to operate with missing links and incomplete or inconsistent information that can be found in the Web.

**There is no need for absolute truth** Not everything found in the Web is true. This introduces the need for the concept of Trust: A Semantic application must decide what resources to trust.

**Support Evolution** Concepts can be defined differently by different people or by the same people at different times. The Semantic Web allows for the concepts to evolve as the human knowledge evolves and expands. It also allows for concepts to be referred using different vocabularies and definitions or add new information without the old being modified.

**Minimalist design** Using protocols with a small and universally understood set of commands and standardizing essential components allows the implementation of applications that are based on already standardized technologies.

The *Explorer's Guide to the Semantic Web* [Pas04] also states some other important features that are described next:

**No state information** Web interactions are stateless. If it is necessary for some business to store information across several interactions, the server must provide the means to make it possible.

**Be as decentralized as possible** The Web is decentralized. If you have a computer on the network, you can put a web server on it; and if you have a server, you can add resources to it without registering them anywhere else.

**Function on a large scale** Independent interactions make possible a large, decentralized system where responses can be cached to allow faster responses and reduce network traffic.

### 2.2.1 A Layered Approach

Figure 2.1 represents the layers of the Semantic Web approach as presented by Tim Berners-Lee in a keynote address delivered at *The Twenty-First National Conference on Artificial Intelligence*.[1] The most significant layers are briefly described here:

**XML** eXtensible Markup Language (XML) has been a standard for exchanging data over the Web in the past years. A XML document contains nested sets of open and close tags, each possibly containing several pairs of attributes and its values.

XML does not define a structure for its contents nor does it provide the means to talk about the semantics of data.

For a more complete understanding of the XML language the reader is referred to [BPSM+06].

**XML Schema** XML Schema (XMLS) is a language that is used to define a grammar for XML documents. This may be necessary since XML has no fixed vocabulary or set of allowable combinations.

**RDF** Resource Description Framework (RDF) is a language designed to describe information and meta data. An alternative is Topic maps, a non-W3C standard. RDF is explained further in Section 2.3 (page 12).

[1] The presentation is available at: `http://www.w3.org/2006/Talks/0718-aaai-tbl/`

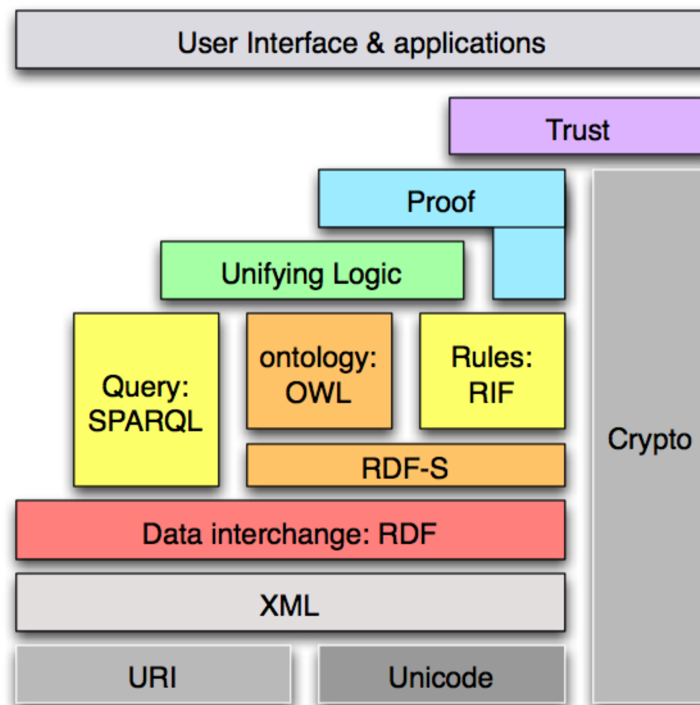


Figure 2.1: A layered approach to the Semantic Web

**RDFS** RDF Schema (RDFS) introduces a type system for RDF models and basic primitives to build ontologies. RDFS supports primitives such as classes, subclasses, subproperties, domain and range restrictions of properties.

It also lets developers define a vocabulary for RDF data and specify the kinds of object to which these attributes can be applied.

**Ontology** RDF Schema is used by many more advanced ontology frameworks such as the Web Ontology Language (OWL), an ontology language designed for the Semantic Web. OWL is further detailed in Section 2.3 (page 15).

**SPARQL** represents another important part of the Semantic Web: querying the stored information. It is a widely used query language for RDF.

**RIF** The Rule Interchange Format (RIF) is an effort to develop a format for the exchange of rules in rule-based systems on the Semantic Web. There is currently available a W3C Recommendation Working Draft [BK07].

**Logic and Proof** Logical reasoning in the Semantic Web is used to determine the consistency and correctness of data. It can also be used to infer conclusions that are not explicitly present in the data.

**Trust** It is necessary to provide means of authentication and establishing trustworthiness of data, services, and agents.

### 2.2.2 Metadata

The Semantic Web proposes that web pages, in addition to containing formatting information to produce a document for human readers, also contain information about their content in a representation easily processable by machines. The term metadata refers to such information: data about data. Metadata is still data, the difference is in the use of the data and, most importantly, in the subject of the metadata: other data.

Metadata information should be expressed using a common format, such as RDF [MM04], in order to enable a faster development and interoperability of Web resources.

For example, the ISBN number and the name of the author name are metadata about a book. Metadata information can be used in searches and in discovering information. It would also allow for users to recommend Web pages that they think are interesting and search engines might take them into account to give results of higher quality.

#### Annotations

Annotations are another form of metadata. To annotate a Web resource means adding information (notes, commentaries, *etc.*) to that resource without changing the original.

There is a distinction between annotations and other metadata: an annotation implies the interaction of a user with a web resource, it results from the perception of a specific user, other than simply representing information about the web resource.

Annotations, although useful for the user, can also provide valuable information if shared across the Web. If a user finds a passage that he thinks is especially important in a web resource, he would have the possibility of highlighting that passage (and possibly adding a comment) so that, next time he visits that resource, the passage is also highlighted and the comment available. It could also be possible to make the annotation available to anyone that visits the resource or a user could enable viewing annotations posted by people he *trusts*.

**Annotea** [Koi07, KK01] is an experimental annotation system available, developed by the W3C that uses RDF to describe the annotations. The W3C also supplies an experimental browser, Amaya [Vat07] testbed for all W3C experiments and validations, that can read and write Annotea annotations. These annotations can be stored either locally in the computer of the user (in this case they are not sharable) or on an Annotea server.

### 2.2.3 Ontologies

Ontologies are an important aspect of the Semantic Web. A possible definition for an ontology is given in [Gru93]:

> "A body of formally represented knowledge is based on a conceptualization: the objects, concepts, and other entities that are presumed to exist in some area of interest and the relationships that hold them. A conceptualization is an abstract, simplified view of the world that we wish to represent for some purpose. (...) An ontology is an explicit specification of a conceptualization."

This definition focuses two important aspects of ontologies: that the description is formal and thus permits easy handling by a computer program and that a specific ontology is designed for some particular domain.

The term ontology originated in philosophy, where it represents the study of the nature of existence. In computer science, more specifically knowledge-based systems, what *exists* are the elements that can be represented.

An ontology is composed of classes and relations between these classes. Classes correspond to important concepts of the domain the ontology represents. Relations are for instance hierarchies of classes. Ontologies can also include information such as properties, restrictions and equality or difference statements between resources.

In the Semantic Web, ontologies provide understanding of a domain, allow to overcome differences in the terminology of the same concept or correct the problem of using identical names with different meanings. Ontologies are also useful for improving the accuracy of Web searches, allowing search engines to search for resources that refer to a specific concept in an ontology. Further details about improving web searches are specified in Section 2.5.1 on page 18.

#### Ontology reasoning

Some possible uses of reasoning in ontologies are stated in [BHS05]:

> "Reasoning is important to ensure the quality of an ontology. (...) During ontology design, it can be used to test whether concepts are non-contradictory and to derive implied relations. (...) [When searching annotated Web pages] interoperability and integration of different ontologies is also an important issue. Integration can, for example, be supported by asserting inter-ontology relationships and testing for consistency and computing the integrated concept hierarchy. Finally, reasoning may also be used when the ontology is deployed, i.e., when a Web page is already annotated with its concepts. One can, for example, determine the consistency of facts stated in the annotation with the ontology or infer instance relationships"

The "*Semantic Web Primer*" [AvH04] indicates some of the forms of reasoning that are possible to be made with ontologies:

**Class membership** If $x$ is an instance of a class C, and C is a subclass of D, one can infer that $x$ is an instance of D.

**Equivalence of classes** If class A is equivalent to class B, and class B is equivalent to class C, then A is also equivalent to C.

**Classification** If certain property-value pairs are declared to be a sufficient condition for membership in a class A and an individual $x$ satisfies such conditions, $x$ can be concluded to be an instance of A.

## 2.3 Web Languages

Next are presented some of the available languages whose main focus is the World Wide Web and ultimately were used as the basis for the ontology language OWL.

### Resource Description Framework

The Resource Description Framework (RDF) [MM04] is a W3C recommendation that aims at standardizing the writing and use of metadata in Web resources. RDF is also the base for other ontology languages.

RDF uses a simple data model consisting of resources (identified by URIs) and statements that can be made about resources. A statement is a triple pattern in the form of object, attribute and value (the attribute can also be called property). This indicates a relation between two resources: the

object and value. The object is a resource or a blank node, the attribute is a resource and the value may be a resource, blank node, or a literal (literals are simple values, like numbers or strings).

RDF can use different forms of syntax: it has a standard syntax using XML but it can also be presented, for instance, using the Turtle syntax. The example of Turtle syntax shown in Figure 2.3 (page 14) corresponds to the Turtle translation of the example in Figure 2.2.[2]

```
1  <?xml version="1.0"?>
2  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3           xmlns:dc="http://purl.org/dc/elements/1.1/"
4           xmlns:ex="http://example.org/stuff/1.0/">
5    <rdf:Description
6              rdf:about="http://www.w3.org/TR/rdf-syntax-grammar"
7              dc:title="RDF/XML Syntax Specification (Revised)">
8      <ex:editor>
9        <rdf:Description ex:fullName="Dave Beckett">
10         <ex:homePage rdf:resource="http://purl.org/net/dajobe/" />
11       </rdf:Description>
12     </ex:editor>
13   </rdf:Description>
14 </rdf:RDF>
```

Figure 2.2: RDF XML syntax example

### Anonymous resources

Anonymous resources (also known as *blank nodes* or *b-nodes*) are resources that have no name. Blank nodes are considered to be unique nodes that can be used in one or more RDF statements allowing to model complex data sets without creating unnecessary identifiers.

### Topic Maps

Topic Maps are an alternative standard for representing information. They were originally developed to handle automated indexing systems and

[2]These examples are available in `http://www.dajobe.org/2004/01/turtle/`

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix ex: <http://example.org/stuff/1.0/> .

<http://www.w3.org/TR/rdf-syntax-grammar>
  dc:title "RDF/XML Syntax Specification (Revised)" ;
  ex:editor [
    ex:fullname "Dave Beckett";
    ex:homePage <http://purl.org/net/dajobe/>
  ] .
```

Figure 2.3: RDF Turtle syntax example

tables of contents but have evolved to handle a wide range of information resources.

The standard for Topic Maps [ISO02] describes the organization of topic maps and defines XML Topic Maps [Top01], a XML-based language for exchanging topic maps.

## RDF Schema

RDF Schema (RDFS) [BG04] allows the association of semantics to a domain. By defining classes, properties and relations it is possible to limit the expressivity of RDF to statements that are coherent in the domain it is representing. This is not possible to achieve using only RDF.

A class defines groups of objects with common characteristics, objects that can be viewed as a set. The relationship between instances (elements of a class) and classes is done in RDF. An important use of classes is to impose restrictions in order to disallow certain statements to be made in the RDF document that uses the schema. It is possible to impose restrictions on the values of a property which corresponds to restricting the range of the property. Or restrict the objects to which the property can be applied which is in fact restricting the domain of the property.

### Hierarchies and Inheritance

With classes it is possible to establish hierarchical relations among them: the subclass relation defines this hierarchy. RDFS allows a class to have multiple superclasses so, the subclass relation implies that instances of a

class are instances of each of its superclasses. RDF Schema also defines that instances of a class *"inherit"* properties and restrictions from its superclasses.

Properties are defined globally, i.e., they are not defined in a specific class. This makes it possible to define new properties and assign values for that properties to the elements of an existing class without changing the structure of that class.

**Property Hierarchies** It is also possible to define hierarchical relationships between properties. Stating that property P is a subproperty of property Q is equivalent to stating that the all pairs of resources that are related by P are also implicitly related by Q.

## Web Ontology Language

Although RDFS allows the definition of classes, properties and restrictions on these elements it may not be enough to model all intended situations. W3C identified a number of use-cases where the expressivity provided by RDFS does not suffice [Hef04], for instance, it is not possible to detect inconsistencies using only RDF and RDFS.

As an attempt to standardize an ontology language with more capabilities than RDFS, the W3C developed OWL: Web Ontology Language. OWL is an evolution of DAML+OIL which in turn was the result of merging the European OIL (Ontology Inference Layer) and DAML [DAR07] (DARPA[3] Agent Markup Language).

OWL is defined as three sublanguages (or species): Full, DL and Lite. Their main differences are described next. A more detailed explanation of the OWL language and the different species is available in [MvH04].

**OWL Full** corresponds to the entire OWL language, it allows for the unrestricted use of all the OWL primitives, RDF and RDFS. OWL Full is entirely compatible with RDF and RDFS: a valid RDF document is also a valid OWL Full document.

The problem with OWL Full is that is does not guarantee complete or efficient reasoning support: the full semantics of OWL do not guarantee that a query contains a solution that is decidable in finite time.

**OWL DL** consists of a subset of the OWL Full constructors. It is based on description logic, a proved complete and decidable form of first-order logic.

[3]DARPA stands for Defense Advanced Research Projects Agency of the United States of America

The main purpose of the DL language is to guarantee computational decidability (which is not guaranteed in OWL Full). In order to do this, several restrictions were imposed. These, among other things, restrict the use of OWL constructors as the subject of other constructors. In practice this prevents the ontology developer from changing the meaning of the constructors.

These restrictions allow OWL DL to provide efficient reasoning support with the disadvantage of losing the full compatibility with RDF and RDFS.

**OWL Lite** OWL Lite is an even stronger restriction of the OWL language, adding more restrictions to OWL DL. This way, OWL Lite is a language that is easy to understand for users and also easy to build tools for.

OWL Lite provides the basics for hierarchy construction such as subclasses and property restrictions.

OWL allows the definition of two types of properties: *object properties* and *datatype properties*. Object properties relate instances to other instances and datatype properties relate instances to datatype values (for example text strings or numbers).

Axioms are used to provide information about classes and properties, for example to specify the equivalence of two classes or the range of a property.

OWL uses the classes and properties defined in RDFS. Due to the restrictions of OWL DL and OWL Lite, instances of a class must be individuals (classes cannot be defined as individuals of other classes).

### Closed-World and Unique-Names Assumptions

In the scenario of the World Wide Web, where only partial information may be available, OWL follows the open-world assumption model. This implies that if a statement cannot be proved true, it is not possible to conclude that it is false.

It also assumes that individuals with different names may be inferred to be same individual. This means OWL does not follow the unique-names assumption: the same individual may be identified by more that one identifier. This assumption is also valid for classes and properties.

## 2.4 Query Languages

Another important aspect of the Semantic Web is the ability to retrieve data modeled by the languages described in the previous section.

Query answering on the Semantic Web is a complex process due to some peculiarities of the Web [FHH04]:

- Include several kinds of query-answering services with access to different types of information represented in different formats.
- Different specifications of servers (partial information, performance limitations or the inability to handle the query).
- It may be necessary to query without specifying the knowledge base that shall be used to answer the query.
- The set of notations and surface syntactic forms used on the Web is already large, and various communities have different preferences, none of then universal.

Furthermore, Semantic web querying must take into account the *meaning* that is defined by metadata and has to properly understand and process it.

To achieve this there are several query languages available for RDF. Overviews and comparisons of query languages are presented in [FLB+06, BBFS05, HBEV04]. There are also attempts to develop a query language for OWL: OWL-QL [FHH04].

## SPARQL

SPARQL (SPARQL Protocol and RDF Query Language) is a query language for RDF based on RDQL [Sea04] and SquishQL [MSR02]. It defines both a query language [PS06], a protocol for the query interaction [Cla06] and the output of results in XML [BB06].

SPARQL works as a query language by matching graph patterns against the data source. The graph pattern may include restrictions and other conditions like optional parts, union, nesting of graphs and filtering of the values of the results.

After the selection of the results, several solution modifiers can be applied, for instance it is possible to change the order of the results, limit their number or change the starting element (by applying an offset).

The result of the query can have different forms: selections of bindings for the variables, yes or no answers or construction of other triples. These results can be presented in the XML format defined in [BB06].

A more detailed description of the SPARQL language is presented in Section 5.1 and the semantics and query forms are presented in [Par06, PAG06].

## 2.5 Applications for the Semantic Web

There are many different perspectives on what the general idea of the Semantic Web is. Some of the most significant ideas are now presented:

### 2.5.1 Improving Web search

Much of the success of World Wide Web is due to search engines. Search engines, like Yahoo or Google, currently work on a keyword basis and are able to retrieve the most relevant web pages in each search. One serious problem is that the relevant resources found are of little use if they are among several thousand less relevant, or even irrelevant other resources.

Another problem with the current web searches is their high sensitivity to vocabulary. If the initial search keywords are not the same as those contained in the web documents, the results will not include all the relevant web pages. This can be the case if the web documents and the search query do not use the same terminology.

Also, the required information may not be present in a single web page. In this case, it is necessary to perform several queries and manually gather all the information from each query result in order to retrieve all the information.

Currently there are tools that can interpret the structure of a web document and perform operations on the text. However, there are still sentences that are hard to differentiate in terms of meaning. There are two alternatives to improve the web search. One is to continue, as has been done so far, to develop artificial intelligence techniques to further filter the results. Another is to represent the web content in a form that is more easily processed by machines. It is for the latter that the information brought by the Semantic Web can help.

### 2.5.2 Web as a database

As stated in [SBLH06] one of the motivations of the Semantic Web comes from the possibility of accessing relational databases by exporting them to the Web using a system of URIs (Universal Resource Identifiers). Relational databases are a common way to store information and it is now possible to retrieve information contained in these databases over the Web.

There is also a large amount of information stored in the Web that is not in the form of relational databases. This data, be it in a relational database or web documents, is generally separate and not easily merged. A part of the Semantic Web vision is to unify the description and retrieval of stored data, thus considering the Web as part of a virtual database.

### 2.5.3 Ubiquitous Computing and Web services

Ubiquitous computing is a paradigm of personal computing where the focus is to have the computing power embedded in the daily environment (using small handheld devices and wireless technologies).

Semantic Web technologies can be used in Ubiquitous Computing to provide better service discovery mechanisms. Current discovery systems are based on standardization: the representation of the service and all communication processes must be known before the communication begins.

With the advent of Semantic Web technologies it would be possible to build discovery systems capable of working almost without any prior intervention.

Semantic gadgets, devices capable of semantic discovery [LA03], are devices that can discover and use other services or devices without any human intervention. This is possible using Semantic Web technologies such as RDF, URIs and metadata.

#### Web Services

Also in the context of web services one could use Semantic Web technologies for Web Service discovery. The composition and execution can also be automated by enabling the semantic description of the Web Services.

This would allow keeping the human intervention to a minimum. Currently both the discovery and query of Web Services require human intervention, at least the first time, in order to analyse the structure of the service.

## 2.6 Conclusion

Now that the Semantic Web is getting more focus and attention, new terms and views of the Semantic Web arise. The idea to retain is that it is part of an evolutionary process: evolution of the current web into an also machine-friendly Web.

# Chapter 3

# Contextual Logic Programming

Aristotle, a Greek philosopher that lived between 384 and 322 BC, is commonly considered to be the father of logic for bringing forth a shift in the focus of mathematics: from computation to proof [AvH04, Pas04]. Logic is the study of correct reasoning, that can be used to deduct correct conclusions, as exemplified by the well known syllogism:

```
All men are mortal
Socrates is a man
Therefore, Socrates is mortal
```

This Chapter is structured as follows: Section 3.1 briefly describes Logic Programming and its use in the Semantic Web and Section 3.2 describes an extension to Logic Programming called Contextual Logic Programming. Later, Section 3.3 and Section 3.4 introduce two used frameworks: ISCO and PiLLoW.

## 3.1 Logic Programming

Logic Programming consists of the interpretation of First-Order Logic (or Predicate logic) as a programming language, as explained in [Kow74, EK76]. Logic Programming is widely used in theorem proving, knowledge representation and artificial intelligence.

Prolog is one of the most widely used logic programming languages and some of the basic concepts: facts, queries and variables, are briefly described next (these concepts are further detailed in [SS86]):

**Rules and Facts** describe relations between other facts or state knowledge. These can be called a *logic program*.

**Queries** can be used to retrieve information, i.e., ask if a relation is true.

**Variables** represent an unspecified value and can be used in facts, rules and queries.

**Compound terms** consist of a functor (the name of the term) followed by a finite number of arguments. The arity of a compound term is the number of arguments it contains. It is possible to identify a compound term by the form: `functor/arity`.

#### Query evaluation

A query is evaluated by finding a pattern in the logic program that matches the given query. If such a pattern is found, the query is said to *succeed* and any variables in the query are *unified* with the terms in the pattern, making the variable identical to the term. Otherwise the query will fail and not perform any *unification* (also called *instantiation*).

In the case of a successful query, there may exist other facts that match the query that would perform different bindings for the variables in the query. These bindings are performed by the *backtrack* process: it will return new bindings for the variables based on the new pattern found.

### 3.1.1 Logic Programming and the Semantic Web

In the Semantic Web context, logic programming can be used to overcome problems that may not be possible using only OWL. They are stated in [MHRS06]:

**Higher Relational Expressivity** With OWL it is only possible to model domains whose objects are connected in a tree-like manner. It may be necessary to model other types of relations: for example, the lack of composition constructor in OWL does not allow the definition the "uncle" relation.

**Closed-World Reasoning** The open-world semantics of OWL only allows to answer positive queries, i.e., answer queries about known facts. Facts not present in the database cannot be considered false so it is not possible to answer these queries.

**Modeling Exceptions** OWL does not allow to model exceptions. Exceptions are common in the real world and it may be necessary to model them. In order to enable exception modeling it is usually necessary to use a form of default negation.

## 3.2 Contextual Logic Programming

Contextual Logic Programming [MP93, MP89] (CxLP) intends to address the issue of modularity in Logic Programming using the concept of *units*. A *unit* consists of a set of predicates, combining them under the same identifier: the name of the *unit*. An example of a unit is presented in Figure 3.1 using the syntax of GNU Prolog/CX (taken from [NAD04]).

```
1 :- unit(teacher).
2
3 name(Name):- teacher(Name, _, _).
4 department(Dep):- teacher(_, Dep, _).
5 degree(Deg):- teacher(_, _, Deg).
6
7 teacher(john, computerScience, phd).
8 teacher(bill, computerScience, msc).
```

Figure 3.1: Example unit: `teacher/0`

These *units* can then be combined to arrange a *context*, called an *execution context*. It is in this dynamically created context that a goal will be executed.

### 3.2.1 GNU Prolog/CX

Throughout this work, a specific implementation of CxLP was used: GNU Prolog/CX [AD03]. This implementation introduces the possibility of defining arguments for units. These arguments act as global variables in the unit where they are defined, i.e., every clause in the unit can access them in the same manner as if they were a variable of the clause. These unit arguments can also act as input of information for a unit when creating a context.

Using unit arguments, the unit shown in Figure 3.1 can be rewritten in the form illustrated in Figure 3.2 (presented in [NAD04]). In this unit, the predicates that access the name, department and degree simply access the corresponding unit argument. A new predicate `item/0` is introduced that will instantiate all the units arguments with the facts defined by predicate `teacher/3`.

A context can be generated using any possible combination of the available units. The contexts are constructed using the defined operator of context

```
:- unit(teacher(NAME, DEPARTMENT, DEGREE))

name(NAME).
department(DEPARTMENT).
degree(DEGREE).

teacher(john, computerScience, phd).
teacher(bill, computerScience, msc).

item:- teacher(NAME, DEPARTMENT, DEGREE).
```

Figure 3.2: Example unit: `teacher/3`

extension: ':>' . The use of this operator, generally in the form `U :> G`, extends the current context with unit U and then resolves the goal G in the new context.

Unit arguments can also be used to transparently query data defined in a unit. For instance, the goal

```
?- teacher(bill, D, _) :> item.
```

will instantiate variable D with 'computerScience'.

### Context resolution

Consider the following definition (presented in [NAD04]): "To derive an atomic goal G in a context $u_1 u_2 \ldots u_n$ a search for the smallest i, $1 \leq$ i $\leq$ n, such that G can be derived with a clause of $u_i$, is made. The derivation of the body of that clause is considered in the reduced context $u_i \ldots u_n$."

In a nutshell, when executing a goal $G$ in a context $C$, a CxLP Engine will traverse $C$ looking for the first unit $u$ that contains a definition for $G$'s predicate. $G$ is then executed, as if it were regular Prolog, in a new context $C'$. $C'$ is the suffix of the context $C$ which starts with unit $u$. The body of the clause (if present) is then derived using the reduced context $C'$.

A CxLP context can be represented by a list of units where the empty list (`[]`) represents the empty context.

### Context operators

A goal will be executed in the context that is headed by the first unit that contains a clause for the goal but there are operators defined that will

allow the manipulation the contexts or change the method of resolution.

Some of the operators of GNU Prolog/CX are presented next.[1] These are called *context operators* and enable the modulation of the context as a part of the computation.

**Context extension:** U `:>` G, this operation extends the current context with unit U and then reduces goal G.

**Context switch:** C `:<` G, attempts to evaluate goal G in context C, ignoring the current context.

**Supercontext:** `:^` G, evaluates goal G in the context resulting of removing the top unit from the current context.

**Current context:** `:<` C, unifies C with the current context.

**Calling context:** `:>` C, unifies C with the calling context.

**Lazy call:** `:#` G, evaluates the goal G in the calling context.

## 3.3 ISCO

The ISCO (*Information System COnstruction*) [AN06] programming language aims to be a mediator between the user and relational databases and provides a way to develop and access organizational information systems.

As stated in [AN06], the Prolog implementation used by ISCO, GNU Prolog/CX, also provides the constraint logic programming paradigm, which is a very useful extension to the traditional Prolog programming style in that it allows for problems to be solved by providing *a priori* search-space pruning, through the constraint propagation mechanism. ISCO fully takes advantage of this feature.

An ISCO class consists of a data structure definition equivalent to a table on a database. After an ISCO file with class definitions is compiled by the ISCO compiler, the system will create those tables on the appropriate back end database and generate access predicates to manipulate that structure.

Figure 3.3 shows the definition of some ISCO classes. According to that figure, the ISCO compiler will create the predicates `dictionary/2` and `dictionary_use/3` which can be used to consult the database.

[1]For a more detailed and formal description, the reader is referred to [AD03].

```
mutable class dictionary.
       id:  serial. key.
       name:  text. unique.

mutable class dictionary_use.
       code:            int. key.
       key:             dictionary.id.
       some_field:      text.
```

Figure 3.3: Example class definitions in ISCO

## 3.4 PiLLoW

Another tool that is widely used in this work is the PiLLoW library [CH07, CHV96]. This library, originally designed for Ciao Prolog allows the generation of HTML structured documents, produce HTML forms and its handlers and also to access and parse WWW documents. In this work PiLLoW is being used mainly to generate the XML files in which SPARQL returns its results.

PiLLoW reads a formatted Prolog term and generates the corresponding HTML/XML. The name of the node corresponds to the functor of the term and the properties of the node are specified in a list after the `'$'` character. It is possible to define a list of elements of this same structure as the argument of the predicate. This list will represent the children of the node.

For example the output presented in Figure 3.5 is generated by the term shown in Figure 3.4.

```
node([
     subnode([
              element([x])$[name=element1],
              element([y])$[name=element2]
     ])
])$[property1=a,property2=b]
```

Figure 3.4: PiLLoW term example

```
<node property1="a" property2="b">
  <subnode>
    <element name="element1">x</element>
    <element name="element2">y</element>
  </subnode>
</node>
```

Figure 3.5: PiLLoW output

## 3.5 Conclusion

This chapter introduced the notion of Contextual Logic Programming (CxLP), necessary for the reader to understand the technology this work is based on, along with some of the concepts and keywords often used to describe it.

# Chapter 4

# The XPTO system

The main objective of the XPTO[1] (XPTO Prolog Translation of Ontologies) system is to represent Web ontologies from the perspective of Contextual Logic Programming.

As mentioned in Section 2.3 on page 15, Web ontologies can be represented using the OWL language and OWL is sub divided into three sub languages: OWL Lite, OWL DL and OWL Full. XPTO is capable of parsing and representing ontologies described in OWL (Lite and DL sub languages). OWL DL emerged as the target for the mapping and representation capability of XPTO since it guarantees computational completeness and decidability, i.e., all conclusions are computable and will finish in finite time [AvH04]. This is not guaranteed by the OWL Full language.

In XPTO the information represented in the ontology is translated into GNU Prolog/CX predicates and units. This process is performed in two phases: the ontology parsing and the unit generation.

During the first phase, the ontology is parsed as a plain XML structure, resulting in a Prolog term representing the complete ontology. This process is described in Section 4.1.

In the unit generation phase, the Prolog term is transformed into a dictionary, an incomplete structure annotated with the necessary information for the generation of the units. Subsequently the unit files are created and loaded into the running instance of the program. Section 4.2 details this process.

In this section the ontology examples are taken from the Wine ontology [W3C06] and the queries are also performed over this ontology.

[1]This work was developed in cooperation with Cláudio Fernandes.

This chapter presents the XML parser used in Section 4.1. Section 4.2 describes the internal representation of ontologies and Section 4.3 introduces the manner in which to retrive the information from the representation. Some use cases for the developed system are presented in Section 4.4 and, finally, Section 4.5 presents the results of the benchmarks performed.

## 4.1 Parsing an ontology

The first step towards building the ontology representation is parsing the ontology file. The parser must be able to read an ontology from a document and represent it in an adequate data structure.

In this phase the ontology is handled as a plain XML file and read in using an available XML parser. To achieve this, several XML parser libraries were considered (mostly Prolog and C parsers and, for benchmark purposes, parsers in other languages such as Java, Python and Caml). The results of these benchmarks are presented in Section 4.5.

The selected parser was the Expat XML parser [Coo06]. The main reasons that influenced the choice of this parser were the results of the benchmark tests and the easy integration of C and Prolog. These reasons are further explained in Section 4.5.1.

The Expat library parses the XML by matching patterns in the text. This way the parser incrementally creates a data structure representing the XML. Once the end of the file is reached, a term is generated based on the created structure and returned to Prolog. This term is an accurate representation of the XML file: apart from any possible comments in the XML file, there is no further loss of information in this transformation.

### Prolog representation for XML

The internal Prolog representation used for a XML structure is a list of `XmlElement`, where an `XmlElement` is a term of the following form:

```
node(ElementName, ElementAttributesList, ElementChildList).
```

This representation will produce the structure represented on Figure 4.2 (page 30) for the XML code in Figure 4.1 (page 29). Each part of the structure is detailed below:

**ElementName** represents the name of the XML element and is stored as an atom or, for URIs, a compound term whose functor is `'#'` and contains the URI and local part as arguments. In the case of the XML element

name does not contain the URI part, the URI will be the empty atom: ''.

This simplifies the handling of these elements within Prolog since it is possible to access each part of the element directly.

**ElementAttributesList** is a list of the attributes of the XML node in the form `AttributeName = AttributeValue`.
`AttributeName` and `AttributeValue` will be represented in the same form as `ElementName`.

**SubElementsList** is a list off all nodes that are exactly one level below in the same branch of the XML document structure. These may be other nodes (elements of the same structure) or element values which will be represented only by the value.

```
<!DOCTYPE rdf:RDF [
          <!ENTITY xsd  "http://www.w3.org/2001/XMLSchema#" >
          ]>
<rdf:RDF xmlns:xsd = "http://www.w3.org/2001/XMLSchema#">
  <owl:Class rdf:ID="Vintage">
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty rdf:resource="#hasVintageYear"/>
        <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">
          1
        </owl:cardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>
</rdf:RDF>
```

Figure 4.1: XML example: OWL Class definition

## 4.2 Representation of an ontology

XPTO is prepared to translate ontologies defined in OWL Lite or OWL DL into Prolog. This mapping process must allow for easy access to the information represented in the ontology, using standard Prolog goals.

```
[node(rdf:'RDF', [xmlns:xsd='http://www.w3.org/2001/XMLSchema'],
   [node(owl:'Class', [rdf:'ID'='Vintage'],
      [node(rdfs:subClassOf,[],
         [node(owl:'Restriction',[],
            [node(owl:onProperty,
               [rdf:resource= #('',hasVintageYear)],
               []),
             node(owl:cardinality,
               [rdf:datatype=#('http://www.w3.org/2001/XMLSchema',
                                 nonNegativeInteger)],
               ['1']
               )]
            )]
         )]
      )]
   )]
```

Figure 4.2: Prolog XML Representation of the example in Figure 4.1

After parsing, the entire ontology is represented by a Prolog term, as explained in Section 4.1. The work at hand is now to generate a dictionary with the necessary information to later create the GNU Prolog/CX units. These generated units are then compiled and loaded into the running program.

The next sections describe the process of translating the Prolog term into the incomplete structure and present the representation of the ontology using GNU Prolog/CX units and predicates.

### 4.2.1 Ontology representation

A GNU Prolog/CX unit is a named and possibly parametrised set of Prolog predicates (as described in Section 3.2). In XPTO, ontologies are represented using *units* and these will be used to represent each OWL class and property, the individuals and one unit containing information about the ontology. This schema is represented in Figure 4.3.

The information about the ontology is represented in a unit with the name `ontologies`. It lists the namespaces, headers, classes and properties of each loaded ontology.

Each class and property is defined in a unit named after the class or property and contains the information that is defined in the ontology about the element. This naming schema for the units of properties and classes does

not present a problem in OWL DL since, as stated in [SWM04], there could never exist a class with the same name as a property:

> "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. This means that, for example, a class cannot be at the same time an individual."

The ontology individuals are represented in the unit `individuals`. It contains the name of the individuals, individual relations and class memberships.

The following sections describe the structure of these units and also discuss some alternative representations previously experimented.

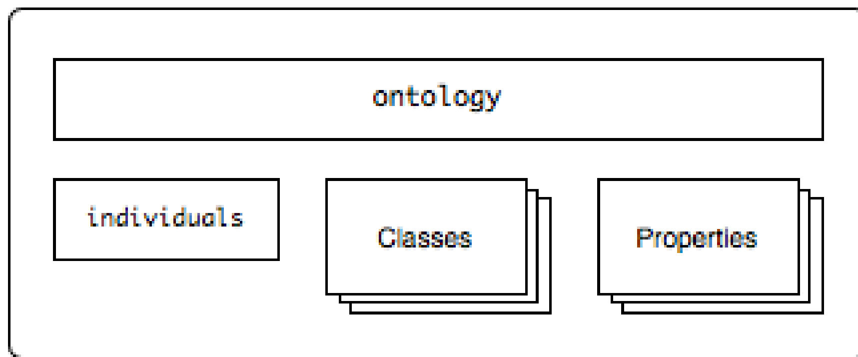


Figure 4.3: Ontology representation schema: units

### `ontologies` Unit

This unit represents the ontology information: XML namespaces, ontology headers, classes and properties. This is done by defining predicates for each case: `ns/3`, `header/3`, `class/2` and `prop/2`. Each predicate contains, in the case of headers and namespaces, an entry with the ontology name, the respective "abbreviation" and value and, for classes and properties, simply the ontology name and the class or property name. The ontology name is included in these predicates to allow the possibility of representing several ontologies.

### Property Units

Each property unit contains the information relative to a specific property. The type of the property (datatype or object) and, if specified any other information such as domain and range, property inheritance and property relations.

These properties also define the method to access their value for specific individuals, that shall be previously retrieved from the context. The way to perform queries on the representation of the ontology is described in Section 4.3.

For example, the definition of a property and the representation are shown, respectively, in Figure 4.4 and Figure 4.5. An example of its usage is shown if Figure 4.11 on page 40.

```
1 <owl:ObjectProperty rdf:ID="locatedIn">
2   <rdf:type rdf:resource="&owl;TransitiveProperty" />
3   <rdfs:domain
4        rdf:resource="http://www.w3.org/2002/07/owl#Thing" />
5   <rdfs:range rdf:resource="#Region" />
6 </owl:ObjectProperty>
```

Figure 4.4: OWL property definition

```
1 :- unit(locatedIn).
2
3 object(rdf : type('TransitiveProperty')).
4 domain('Thing').
5 range('Region').
6 type(object).
```

Figure 4.5: Prolog representation of Figure 4.4

### Class Units

These units will represent each class of the ontology and all information relevant to it; this includes restrictions on the individual properties (class membership) and class inheritance.

It also includes a predicate `class_name/1` that provides the name of the current class. This predicate is used in by the query engine to determine the class that the query refers to as described in more detail in Section 4.3.

**Unnamed classes** These are classes defined implicitly by a set of individuals. They are represented internally by a unit (in the same manner as a named class) but, since they are not assigned a name in the ontology, one is generated for them. This generated name consists of the prefix `___class_` followed by a sequential number.

An example of the use of unnamed classes, using an enumeration, is shown in Section 4.2.2.

### `individuals` Unit

This unit contains all the individuals, their properties and information about individual relations. The individual properties are stored as triples, in the manner of RDF, defined in the predicate `property/3`. The first argument of this predicate indicates the name of the individual, the second corresponds to the property and the third argument contains the value of the property for that individual.

Class membership is defined in the predicate `individual_class/2`. This predicate lists all the individuals, along with their class. Individuals from unnamed classes are not included in this listing: they are only present in the unit that represents the class. This is done to avoid unwanted repetitions when querying the ontology that would be generated if the individuals of the unnamed classes were listed as the other individuals. These individuals are only available in the predicate `individual/1` present in each unnamed class unit.

**Individual relations** In this unit there are also predicates that represent the individual relations, such as `differentFrom/2` and `sameAs/2`, each with individual names as their arguments. These indicate, respectively, that the referred individuals are different or the same [MvH04]. The constructor `owl:AllDifferent` is represented as several `differentFrom` statements, each individual present in the constructor will generate one `differentFrom` statement relating it to every other individual in the list. This is detailed in Section 4.2.2.

**Alternative representations**

To achieve the described representation of the ontology two other representations were tested. The representations described next are the ones that were explored and other solutions may exist:

1. One approach to map an ontology is to represent each property and class in the ontology as a unit and represent the individuals as an instantiation of a goal of the unit that represents the class.

   The units that represent each class have their arity determined by the number of properties defined in the ontology and one extra argument to represent the individual name. This extra argument is referred internally with the name "id" and thus any query asking for the argument "id" will match the name of an OWL individual. An example of a class with this structure is shown in Figure 4.6.

   The individuals of a class are stored in a unit named `ClassName_owl`. This unit contains the instantiations of the predicate `individual/1` defined in the class unit.

   The call to the predicate `item/0` (or `item/1`) instantiates the units arguments. The `item/1` goal returns a compound term that represents the individual and the value of the properties for that individual. The `item/0` allows accessing the individual properties by their name and enables to select only some properties to be shown by using the predicates defined in the unit that access the correct property (as presented in Figure 4.6).

   In this representation, the fixed arity of the representation for the individuals was not appropriate as some individuals may not have a value for all the properties and may have values for properties that are not present in the representation. The solution would be to include all the properties of the ontology in the representation of the individuals. This would implicate an arbitrarily large number of arguments in the class units arguments (equal to the number of properties defined in the ontology).

2. Another representation would be to include in each class unit a list of the names of their individuals, defined in the predicate `individual/1`. Each individual would also be represented in a unit named after the name of the individual. The class unit also defines the method to access the individuals: `item/1` and `item/2`. The first predicate instantiates the argument, by backtracking, with the name of each individual that

```
:- unit('IceWine'(A,B,C)).

% access-predicates
item:- item(_).
item('IceWine'(A,B,C)) :- individual('IceWine'(A,B,C)).

% properties
id(A).
hasBody(B).
hasFlavor(C).
```

Figure 4.6: `IceWine` class unit (first version, partial)

belongs to the class. The `item/2` predicate also instantiates its second argument with a list of all the properties of the individual. The definition of these predicates is illustrated in Figure 4.7.

The representation of each individual in a separate unit could pose a problem as the number of individuals increases, both in terms of representation and querying. In terms of representing this would cause the number of generated units to be very large and cause the compilation process to take long amounts of time.

```
item(A) :-
        individual(A).
item(A, B) :-
        individual(A),
        findall(C = D, individuals :> property(A, C, D), B).
```

Figure 4.7: Class `item` goals example

### 4.2.2 Name analysis

The next process in the loading of ontologies consists in building a dictionary with all the information necessary to generate the units and predicates that will represent the ontology.

The dictionary is implemented as an incomplete structure in Prolog. It is split into four sections: `ontology`, `classes`, `individuals` and `properties`. The properties and classes sections are each a dictionary where the key is

the name of the element at hand. The ontology entry stores information about the ontology, i.e, the info expressed in the `owl:Ontology` node; finally the `individuals` entry stores all the information about individuals. The information about individuals is also grouped by the predicates defined in the individuals unit (`individual_class`, `property`, `differentFrom` and `sameAs`) as previously described (Section 4.2.1).

The term that represents the ontology is parsed according to the specifications of the OWL language as detailed in [MvH04]. Next are presented some of the representation choices that were made.

### Enumeration

An enumeration can be defined as an anonymous class that is defined by a set of individuals and is used, for instance, with the `AllValuesFrom` constructor as represented in Figure 4.8. Classes like this are represented internally like any other OWL class and, in order to do this, they are assigned an internal name (that consists of the prefix `___class_` followed by a sequential number). The individuals of these classes are listed directly in the unit that represents the class and are not present in the `individuals` unit (as explained in Section 4.2.1).

```
1 <owl:allValuesFrom>
2   <owl:Class>
3     <owl:oneOf rdf:parseType="Collection">
4       <owl:Thing rdf:about="#CheninBlancGrape" />
5       <owl:Thing rdf:about="#PinotBlancGrape" />
6       <owl:Thing rdf:about="#SauvignonBlancGrape" />
7     </owl:oneOf>
8   </owl:Class>
9 </owl:allValuesFrom>
```

Figure 4.8: `AllValuesFrom` example

### owl:AllDifferent

The `owl:AllDifferent` constructor indicates that all the individuals it lists are different from each other and, as stated in Section 4.2.1 (see page 33), is represented as several `differentFrom` statements. This is done to simplify

the representation and computation by having only one representation for the same type of information.

For each individual present in the `owl:AllDifferent` list, are generated `owl:differentFrom` facts relating it to every other individual that comes after it in the list. Since the constructor `owl:differentFrom` is symmetric, this will relate all the individuals between them without generating redundant information. For instance, the element in Figure 4.9 will generate the facts represented in Figure 4.10 in the unit `individuals`.

```
1 <owl:AllDifferent>
2   <owl:distinctMembers rdf:parseType="Collection">
3     <vin:WineColor rdf:about="#Red" />
4     <vin:WineColor rdf:about="#White" />
5     <vin:WineColor rdf:about="#Rose" />
6   </owl:distinctMembers>
7 </owl:AllDifferent>
```

Figure 4.9: `AllDifferent` example

```
1 differentFrom('Red', 'White').
2 differentFrom('Red', 'Rose').
3 differentFrom('White', 'Rose').
```

Figure 4.10: `AllDifferent` representation

### Document Checker Conformance

The W3C defines [CR04] what actions a OWL document checker should do. As a syntax checker, it should receive a document as input and identify it as belonging to a specific OWL specie (Lite, DL or Full) or Other if it does not correspond to any of the species.

XPTO performs some consistency checks that are described next:

- A check that is done is to validate the types of properties: in OWL DL a property cannot be subproperty of another that is not of the same type e.g., a `DatatypeProperty` cannot be subproperty of an

`ObjectProperty` and vice-versa. This consistency is achieved by using *type inference*.

- Another test that is performed is to ensure that only constructors allowed by the selected OWL variant are used, for example, it is not possible to use `owl:hasValue` in OWL Lite.

**Namespaces and Annotations**

Annotations are textual notes that can be defined and used within OWL documents. There are five annotation properties predefined by OWL:

- owl:versionInfo
- rdfs:label
- rdfs:comment
- rdfs:seeAlso
- rdfs:isDefinedBy

OWL DL allows annotations on classes, properties, individuals and ontology headers, but only under certain conditions described in [DSB+04]. Annotations are currently being discarded by XPTO. One possible representation for the annotations would have been to define a predicate `annotation/1` in the unit of the element that the annotation corresponds to.

Within the ontology headers are the namespaces. They provide a method of unambiguously interpreting identifiers and making the rest of the ontology presentation more readable. The namespaces of the ontology are being stored by XPTO in the `ontology` unit as described in Section 4.2.1, however the namespaces are currently not being returned along with the solutions to a query, i.e., the solutions are not URIs and are identified only by the name or value of the element.

### 4.2.3 Unit generation and loading

After parsing the ontology and performing semantic analysis to achieve an annotated representation of the ontology, the information required to build the formal representation of the ontology is entirely available. The process of loading the representation of the ontology can be disassembled into three distinct steps:

**Unit generation:** The first step is to generate all the unit files. For each symbol in the dictionary, a unit with the same name as the symbol is generated.

**Compilation:** In order to be loaded into the running program, each unit must be compiled using the GNU Prolog/CX compiler. This means the system, after parsing an ontology and generating the units, must compile every Prolog file that contains a generated unit.

**Loading:** After all the units have been compiled they are ready to be loaded into the program. This is done using the *dynamic loading* of GNU Prolog/CX. Loading each compiled unit makes the ontology representation fully integrated with the running program.

## 4.3 Querying an ontology

At the end of the representation process the ontology is available to be queried using the regular GNU Prolog/CX environment. The way to query the ontology is to build a context using the units that represent the properties and calling the goal `item/0` to activate the query resolution. The query must be prefixed with the `'/>'` operator and optionally a class unit. Other units, described in 4.3.1, can be placed in the context to add further query capabilities or be used as a filter for the results.

For convenience purposes there is also available the goal `item/1`. This goal will instantiate its argument, by backtracking, with the names of the individuals that match the query. This is explained further in Section 4.3.1.

By placing a class unit before the operator `'/>'` it is possible to access only the individuals of that class, or all the individuals of the ontology if the operator is used alone. Querying property values can be achieved by adding to the context the unit that represents the property (Figure 4.11) or by the inclusion of the unit `property/2` to access a value without knowing the name of the property (as shown in Figure 4.12).

The responsibility of setting up a complete query context lies with the `'/>'` operator, it places the `individuals/0` and `access/0` units in the context. For example, for the query present in Figure 4.11, the complete context is shown in Figure 4.13. The `individuals/0` unit is the unit that contains the individuals and property values. The unit `access/0` (partially represented in Figure 4.14) is responsible for accessing the individuals of the ontology, or of a specific ontology class, and instantiating the argument of the `item/1` goal with the individual name. This is the individual that will be used by the other units in the context.

```
| ?- 'IceWine' /> locatedIn(L) :> hasFlavor(F) :> item(I).

F = 'Moderate'
I = 'SelaksIceWine'
L = 'NewZealandRegion' ?
```

Figure 4.11: ontology query (direct access)

```
| ?- 'IceWine' /> property(locatedIn,L) :>
                  property(F,'Moderate') :> item(I).
F = hasFlavor
I = 'SelaksIceWine'
L = 'NewZealandRegion' ?
```

Figure 4.12: ontology query (`property/2` example)

There is also the possibility of defining custom predicates that use this operator in order to be used by a Prolog programmer (this is presented in Section 4.3.2).

### 4.3.1 Units for refining ontology queries

There are also available some units that can be used in the query to retrieve other values or perform some operations. They are described next:

**individual/1** Including this unit in the context unifies the argument of the unit with the individual name. Using this unit provides a explicit query form, by querying the individual name and calling the goal `item/0`. It is also possible to query the individual name by using the `item/1` goal.

```
| ?- individuals :> access :>
     'IceWine' :> locatedIn(L) :> hasFlavor(F) :>
     item(I).
```

Figure 4.13: Example of a complete query context

```
:- unit(access).

item(A) :-
        :# class_name(CL),   % check if there is a class
        individuals(CL, A).  % in the context and get the elements

individuals(CL, I):-
        individual_class(I, CL).     % elements of the class
individuals(CL, I):-
         [CL] :< superClassOf(C),    % elements of the subclasses
         individuals(C, I).
```

Figure 4.14: unit `access/0` (partial)

```
| ?- /> individual(I) :> item.
I = 'WhitehallLanePrimavera' ?
```

**class/1** If this unit is included in the context it will unify its argument with the class of the matching individual. This is useful to determine the class of the individual when querying the entire ontology. This also allows to restrict the results of the query to a specific class, i.e, not including the individuals of the subclasses, as is the default behaviour when including the class unit before the `'/>'` operator.

```
| ?- /> class(C) :> item(I).
C = 'DessertWine'
I = 'WhitehallLanePrimavera' ?
```

**property/2** This unit allows to access the properties of the individual without prior knowledge of the property name or to query the property name based on the property value. The first argument is the property name and the second the property value.

```
| ?- 'IceWine' /> individual(I) :> property(P,V) :> item.
I = 'SelaksIceWine'
P = locatedIn
V = 'NewZealandRegion' ?
```

**all/2** Including this unit in the execution context is analogous to using a `findall` in Prolog. The first argument is the element structure and the second will be the list of the elements in the specified form. This allows to retrieve the set of solutions for the variables present in the query.

```
| ?- 'Chardonnay' /> individual(I):> all(I, L) :> item.
L = ['BancroftChardonnay',
     'FormanChardonnay',
     'MountEdenVineyardEdnaValleyChardonnay',
     'MountadamChardonnay',
     'PeterMccoyChardonnay']
```

**optional/1** This unit receives as argument another unit such as `property/2` or a property unit and will succeed with the results if the specified unit succeeds. Otherwise it will succeed leaving any variables in the argument unbound. This is similar to the SPARQL `optional` statement [PS06].

### 4.3.2 Native Prolog query representation

To make simple queries easier for Prolog programmers, custom predicates can be created to encapsulate the contextual queries. The arguments to these predicates must be defined explicitly after loading the ontology and follow the conventions:

- The predicate functor is the name of the class
- The first argument is the name of the individual.

The arguments that are present in the predicate after the individual name are specified when defining the predicates. This specification requires indicating the class for which to generate the predicate (that will be the functor of the predicate) and a list of properties that corresponds to the sequence of arguments after the *individual* as shown for example in Figure 4.15. This allows the user to choose which properties will be present in the generated predicate. The generated Prolog representation is listed in Figure 4.16.

This approach is limited due to the fixed arity of the predicates. Individuals may not have a value for some of the properties (an unbound variable for that property will be returned in this case) and may contain properties that are not present in the predicate.

```
pred('IceWine',[hasMaker,hasColor])
```

Figure 4.15: predicate definition example

```
'IceWine'(A, B, C) :-
        'IceWine' /> optional(hasMaker(B)) :>
                     optional(hasColor(C)) :>
                     item(A).
```

Figure 4.16: generated predicate

It does, however, conform to standard Prolog programming practice, by allowing the use of positional arguments. It is also possible to define, for each class, several predicates with different arities each containing different properties to be queried.

## 4.4 Example Use Cases

In this section some use case examples for XPTO are presented. First, XPTO queries are compared with SPARQL[2] queries in terms of expressiveness. Then a possible scenario is presented, in which ontology data access using XPTO is merged with database access using ISCO [AN06].

### 4.4.1 SPARQL Query examples

Next are shown some SPARQL query examples and the corresponding query performed using XPTO query syntax. These examples queries are taken from the SPARQL examples of [BBFS05].

#### Query 1

This query is meant to show the selection and extraction capabilities of SPARQL and the intended meaning is stated to be: "Select all Essays together with their authors (i.e. author items and corresponding names)" (Figure 4.17). The corresponding query in XPTO is shown in Figure 4.18.

[2]The SPARQL language is described in Section 5.1.

In XPTO, the `SELECT` statement has no direct representation, it is implicitly defined by the Prolog variables present in the query. As stated in Section 4.2.2, the namespaces are currently being ignored.

```
1 PREFIX   books: http://example.org/books#
2 PREFIX   rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#
3 SELECT   ?essay, ?author, ?authorName, ?translator
4 FROM     http://example.org/books
5 WHERE    (?essay books:author      ?author),
6          (?author books:authorName ?authorName)
7 OPTIONAL (?essay books:translator ?translator)
```

Figure 4.17: SPARQL example: Query 1

```
1 | ?- /> author(AUTHOR) :> item(ESSAY),
2      /> authorName(AUTHORNAME) :> item(AUTHOR),
3      /> optional(translator(TRANSLATOR)) :> item(ESSAY).
```

Figure 4.18: Query 1 - XPTO syntax

### Query 4

Query 4 (Figure 4.19) is: "Invert the relation author (from a book to an author) into a relation authored (from an author to a book)."

This query intends to show the SPARQL ability to return RDF triples using the `CONSTRUCT` statement. The developed system does not directly address this, it allows only variable binding queries. In order to return the desired structure it would have to be done explicitly, using additional Prolog goals. The query that returns the data necessary is shown in Figure 4.20. The use of the `individual/1` unit has the same effect as using the `item/1` goal.

### Query 9

Query 9 is stated as: "Return the co-author relation between two persons that stand in author relationships with the same book." (Figure 4.21). The query using XPTO query syntax is shown in Figure 4.22.

```
PREFIX   books: http://example.org/books#
CONSTRUCT (?y books:authored ?x)
FROM      http://example.org/books
WHERE     (?x books:author ?y)
```

Figure 4.19: SPARQL example: Query 4

```
| ?- /> author(Y) :> individual(X) :> item,
     I = authored(X,Y).
```

Figure 4.20: Query 4 - XPTO syntax

### 4.4.2 Database integration using ISCO

One important aspect of the Semantic Web vision is the aim for sharing and open data access. Data can be provided from different sources and formats. In this section is demonstrated, with an example about the Periodic Table[3], how to write a Prolog program that can, using XPTO and ISCO [AN06], query different data sources, namely ontologies and databases.

#### The Periodic Table

For example purposes, two sources of information about the periodic table are used. One will be an ontology[4] that describes the main components

[3] A periodic table to use as a reference can be found at http://www.webelements.com/

[4] The OWL representation of the Periodic Table that was used was written by Michael Cook: http://www.daml.org/2003/01/periodictable/

```
PREFIX    books: http://example.org/books#
CONSTRUCT (?x books:co-author ?y)
FROM      http://example.org/books
WHERE     (?book books:author ?x)
          (?book books:author ?y)
AND       (?x neq ?y)
```

Figure 4.21: SPARQL example: Query 9

```
1 | ?- /> author(X) :> item(BOOK),
2      /> author(Y) :> item(BOOK),
3      X \= Y,
4      I = coauthor(X,Y).
```

Figure 4.22: Query 9 - XPTO syntax

of the periodic table as ontology classes, e.g., `Groups`, `Blocks`, `Elements`, etc, and the other a database with detailed information about each element. Combining both, it is possible to access information about the detailed characteristics of the elements that belong to a particular `Group` or `Period`.

### Accessing the data

The definition of the `Group` class in the Periodic Table ontology, contains, among others, the following properties: `number`, `name` and `element`. An individual of this class: `group_10`, is shown in Figure 4.23.

```
1 <Group rdf:ID="group_10">
2     [...]
3     <number rdf:datatype="&xsd;integer">10</number>
4     <element rdf:resource="#Ni"/>
5     <element rdf:resource="#Pd"/>
6     <element rdf:resource="#Pt"/>
7     <element rdf:resource="#Uun"/>
```

Figure 4.23: Periodic table ontology example: `group_10`

Information about the periodic table elements is present in a database that can be defined with ISCO [AN06]. Part of the table `element` definition is illustrated in Figure 4.24.

Now, having both the referred ontology loaded into XPTO and the database accessible via ISCO, it is possible to write Prolog programs to query both data sets. Using the `Group` class defined by the ontology and the `elements` table defined in the database, the following query can be formulated: what is the classification and color of the elements belonging to the group `group_10` (Figure 4.25).

```
mutable class element.
       code:           int. key.
       name:           text. unique
       symbol:         text. unique
       group           int.
       color           text.
       classification  int.
       [...]
```

Figure 4.24: Database table example: `element`

```
| ?-  % access ontology
     'Group' /> element(ELEMT) :>
               number(_NUM) :> item(group_10),

     % access DB using ISCO
     element@(group=_NUM, name=ELEMT,
              classification=CLASSF, color=COLOR).

CLASSF = 'Metallic'
COLOR = 'lustrous, metallic, silvery tinge'
ELEMT = 'nickel'
```

Figure 4.25: Query example (ontologies and ISCO)

Variables `ELEMT` and `NUM` will bind both data sources and, by backtrack, `CLASSF`, `ELEMT` and `COLOR` will return all the solutions available.

## 4.5 Benchmarks

This section presents the performed benchmarks. These include the XML parser, the representation of the ontology and finally, the XPTO representation times are further explained.

### 4.5.1 XML Parsers

Next are presented benchmark results of the parsers tested and the reasons for choosing the XML Expat library are stated.

The files listed in the tables shown are a subset of the files used in the benchmark process. It is an illustrative subset covering several different file sizes, ranging from 400KB to 99MB.

**Test Conditions** The parsers are tested in a dedicated workstation: a Intel Pentium 4 with hyperthread running at 3.2Ghz with 1GB of RAM.

Parse times are measured using the `time(1)` Linux command collecting the `elapsed time`, `system time` and `user time` of 100 runs of the parser. The final average is obtained by removing the 5 worst and best times and calculating the average of the remaining times. As reported by `time(1)` the `system time` represents the number of seconds used by the system in operations for the process, the `user time` is the number of seconds used directly by the process and `elapsed time` corresponds to the real time (total amount of time) used by the process. In order to time only the parse process (not taking into account process allocation times, etc) the average time it takes for each parser to read an empty file is deducted from the parse time of each file.

#### Libxml2, Libexpat1 and Prolog overhead

The Expat XML parser and Libxml2 are two of the available XML parsers written in the C language. The Expat parser is used by the Mozillabrowser and Libxml2 by the Gnome Project [Vei06]. Both parsers were tested in equal environments and in two different situations: as standalone parsers and integrated with Prolog in order to time the overhead of this integration. Table 4.1 and Figure 4.26 show the results obtained. The times labeled as `pl-expat` and `pl-libxml2` are those of each parser integrated with Prolog, respectively Expat and Libxml2.

As Table 4.1 illustrates, in both tested cases the Expat library presents better times that Libxml2.

The times presented in Table 4.1 for `pl-expat` are different from those in Table 4.2 for `pl-expat-v2`: the parser `pl-expat` does not return anything to Prolog (it behaves as expat with the difference that it is called from a Prolog process) but the `pl-expat-v2` parser builds the structures and terms that represent the XML file and returns the term to Prolog. The `pl-expat` parser was tested to time the Prolog overhead whereas the `pl-expat-v2` parser is the parser used in XPTO.

Table 4.1: Libxml2 and Libexpat comparison (seconds)

| File (Size) | Expat | pl-expat | libxml2 | pl-libxml2 |
|---|---|---|---|---|
| file02 (3,5 MB) | 0.04 | 0.06 | 0.07 | 0.08 |
| file03 (1.2 MB) | 0.03 | 0.04 | 0.07 | 0.07 |
| file10 (5.5 MB) | 0.14 | 0.16 | 0.25 | 0.3 |
| file13 (1.6 MB) | 0.04 | 0.05 | 0.06 | 0.07 |
| file17 (24.8 MB) | 0.69 | 0.79 | 2.02 | 2.04 |
| file19 (2.6 MB) | 0.05 | 0.06 | 0.1 | 0.1 |
| file21 (2.3 MB) | 0.05 | 0.06 | 0.09 | 0.1 |
| file22 (14.4 MB) | 0.37 | 0.45 | 0.8 | 0.8 |
| file25 (21 MB) | 0.55 | 0.63 | 0.97 | 0.97 |
| file27 (32.9 MB) | 0.62 | 0.75 | 1.61 | 1.84 |
| file33 (98 MB) | 2.68 | 3.1 | 4.82 | 4.83 |
| file34 (4.5 MB) | 0.12 | 0.15 | 0.25 | 0.25 |

As the results presented in Table 4.1 and Figure 4.26 show, the impact of integrating Prolog with Libxml2 is virtually irrelevant. For Libexpat, on file 33 (98 MB) there is a 15% overhead. On the smaller files, although the overhead percentage remains the same, the impact is also not relevant due to small times measured (under one second).

### Comparison with other parsers

The implemented parser module was benchmarked against other existing XML parsers. The tested are presented next:

**PiLLoW (in GNU-Prolog):** Pillow [GH01] is a web programming library developed at UPM - Technical University of Madrid that provides World Wide Web connectivity for Logic Programming and Constraint Logic Programing systems. It contains a module that implements predicates which generate and parse HTML/XML documents.

**SWI-Prolog:** This is a parser implemented in SWI-Prolog [Wie03], which parses a XML file into a Prolog term. It uses the SWI-Prolog SGML/XML parser, which means it allows for processing partial documents and process the DTD (Document Type Definition) separately.

**W4:** A non-validating parser written in XSB Prolog by Carlos Damásio [Dam07] that produces a Prolog representation of the XML document. It has support for XML Namespaces, XML Base and complying to the recommendations of XML Info Sets.

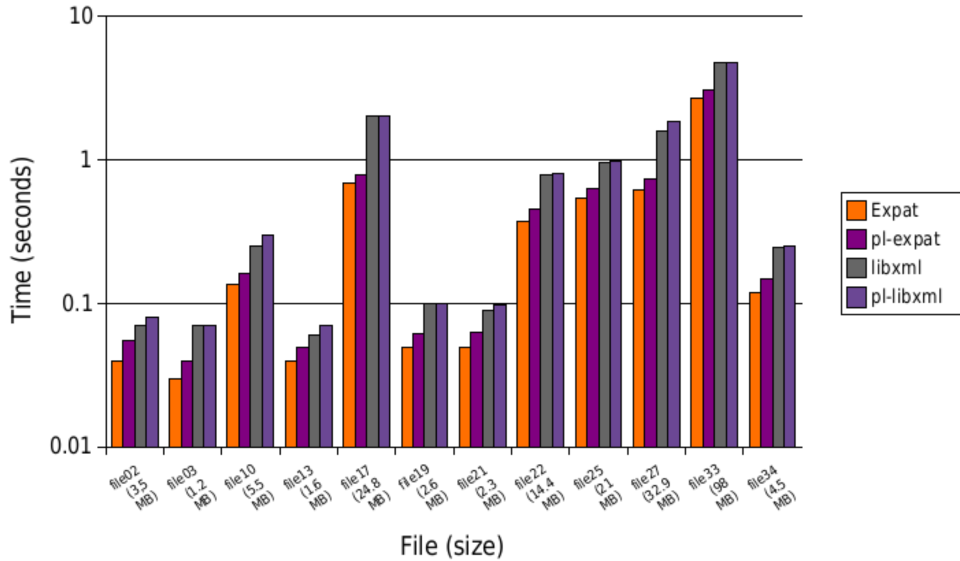


Figure 4.26: Expat Library vs Libxml2

**Jena:** Jena [Jen06] is a semantic Web framework for Java. Among other tools, it has a RDF/XML parser called APR which can be used integrated with Jena or as a standalone parser. Within the framework, two packages were used: one provides a set of abstractions and convenience classes for accessing and manipulating ontologies represented in RDF, and another for creating and manipulating RDF graphs.

**Ciao Prolog:** Ciao [GH99] is a public domain multi-paradigm programming environment, it is a complete Prolog system that allows both restricting and extending the language. Ciao also supports programming with functions, constraints and objects and enables the use of persistence and concurrency. It has a module that implements the predicates of the PiLLoW package related to HTML/ XML generation and parsing.

**OCaml:** Objective Caml [Rém00] is one of the most popular variant of the ML language. It extends the core Caml language with an object-oriented layer and a module system. To parse XML documents, the PXP [Sto07] OCaml library: Polymorphic XML Parser was used.

Performing benchmarks with these parsers enables the comparison of the XPTO parser not only with similar Prolog driven parsers but also with

parsers written in different programming languages and following different paradigms.

Table 4.2 shows all the parse times measured for each of the files and parsers tested. The parser `pl-expat-v2` is the parser used in XPTO.

Table 4.2: Benchmark results (seconds)

| File (Size) | Ciao | pl-expat-v2 | jena | ocaml-pxp | swi | w4 | pillow |
|---|---|---|---|---|---|---|---|
| file02 (3,5 MB) | 3.7 | 0.19 | 2.08 | 1.4 | 0.33 | 9.96 | 2.94 |
| file03 (1.2 MB) | 0.91 | 0.89 | 1.51 | – | 0.19 | 3.55 | 1.15 |
| file10 (5.5 MB) | – | 17.3 | 3.48 | 6.76 | 0.82 | 16.74 | – |
| file13 (1.6 MB) | 1.24 | 0.84 | 2.37 | 1.66 | 0.2 | 3.25 | 1.62 |
| file17 (24.8 MB) | – | – | – | – | 4.29 | 84.25 | – |
| file19 (2.6 MB) | 1.99 | 2.65 | 1.51 | 2.82 | 0.36 | 7.81 | 2.29 |
| file21 (2.3 MB) | 1.67 | 2.2 | 1.58 | 1.66 | 0.3 | 6.63 | 2.05 |
| file22 (14.4 MB) | – | – | 23.03 | – | 2.26 | 45.34 | – |
| file25 (21 MB) | – | – | 10.89 | 25.2 | 3.39 | 63.86 | – |
| file27 (32.9 MB) | – | – | – | 39.03 | 4.75 | 64.97 | – |
| file33 (98 MB) | – | – | – | 127.37 | 17 | – | – |
| file34 (4.5 MB) | – | 73.22 | 3.86 | – | 0.79 | 13.43 | 4.46 |

The results shown in Table 4.2 are from different parsers, not only in terms of programming language but also in terms of features of the parser. The w4 parser performs some validations (in terms of encodings) and represents the whole information in the file. The `ocaml-pxp` is not able to parse some files due to not recognizing statements encountered.

Overall, the `swi` parser revels the best results, both in terms of parse times and number of files parsed. The `pl-expat-v2` parser used in XPTO, as expected cannot handle the larger files, however it presents good results for smaller files (up to 6MB).

**Speedups**

The `speedups` were also calculated, based on the presented results. The purpose is to compare the results using an entity as base and thus relating the others to the chosen one. This enables a better understanding of the results as they are directly compared with a common reference.

Table 4.3 shows the obtained results for the *speedup* calculations. As the base results were chosen the times measured by `Jena` since it is one of the most often used and wide spread parsers. So, every `speedup` is calculated by dividing the `Jena` time by the parser time.

Table 4.3: Speedups results. Jena used as reference.

| File (Size) | Ciao | pl-expat-v2 | ocaml | swi | w4 | pillow |
|---|---|---|---|---|---|---|
| file02 (3,5 MB) | 0.56 | 10.85 | 1.49 | 6.22 | 0.21 | 0.71 |
| file03 (1.2 MB) | 1.66 | 1.69 | – | 7.91 | 0.43 | 1.31 |
| file10 (5.5 MB) | – | 0.2 | 0.51 | 4.22 | 0.21 | – |
| file13 (1.6 MB) | 1.92 | 2.83 | 1.43 | 11.64 | 0.73 | 1.47 |
| file19 (2.6 MB) | 0.76 | 0.57 | 0.53 | 4.17 | 0.19 | 0.66 |
| file21 (2.3 MB) | 0.95 | 0.72 | 0.95 | 5.32 | 0.24 | 0.77 |
| file22 (14.4 MB) | – | – | – | 10.2 | 0.51 | – |
| file25 (21 MB) | – | – | 0.43 | 3.21 | 0.17 | – |
| file34 (4.5 MB) | – | 0.05 | – | 4.91 | 0.29 | 0.87 |

Table 4.4: Speedups results (percentage). Jena used as reference.

| File (Size) | Ciao | pl-expat-v2 | ocaml | swi | w4 | pillow |
|---|---|---|---|---|---|---|
| file02 (3,5 MB) | -44% | 985% | 49% | 522% | -79% | -29% |
| file03 (1.2 MB) | 66% | 69% | – | 691% | -57% | 31% |
| file10 (5.5 MB) | – | -80% | -49% | 322% | -79% | – |
| file13 (1.6 MB) | 92% | 183% | 43% | 1064% | -27% | 47% |
| file19 (2.6 MB) | -24% | -43% | -47% | 317% | -81% | -34% |
| file21 (2.3 MB) | -5% | -28% | -5% | 432% | -76% | -23% |
| file22 (14.4 MB) | – | – | – | 920% | -49% | – |
| file25 (21 MB) | – | – | -57% | 221% | -83% | – |
| file34 (4.5 MB) | – | -95% | – | 391% | -71% | -13% |

Figure 4.27 graphically illustrates the results obtained by the `speedup` calculations. Analyzing the values it is possible to conclude that the parser chosen as reference (Jena) is not the one that presents the best results. The best values are from the `SWI Prolog` parser.

For a better understanding of the results, Table 4.4 shows these represented as percentages. From the calculations it is possible to see that the `pl-expat-v2` parser performs better (in relation to `Jena`) for 3 files, worse times for 4 files and that it can not parse two files that `Jena` can. The largest difference is in `file02`, where the `pl-expat-v2` parser presents a measured time 985% better than `Jena`. For the other results it is also possible to conclude that the times measured by the `SWI Prolog` are always better than `Jena`.

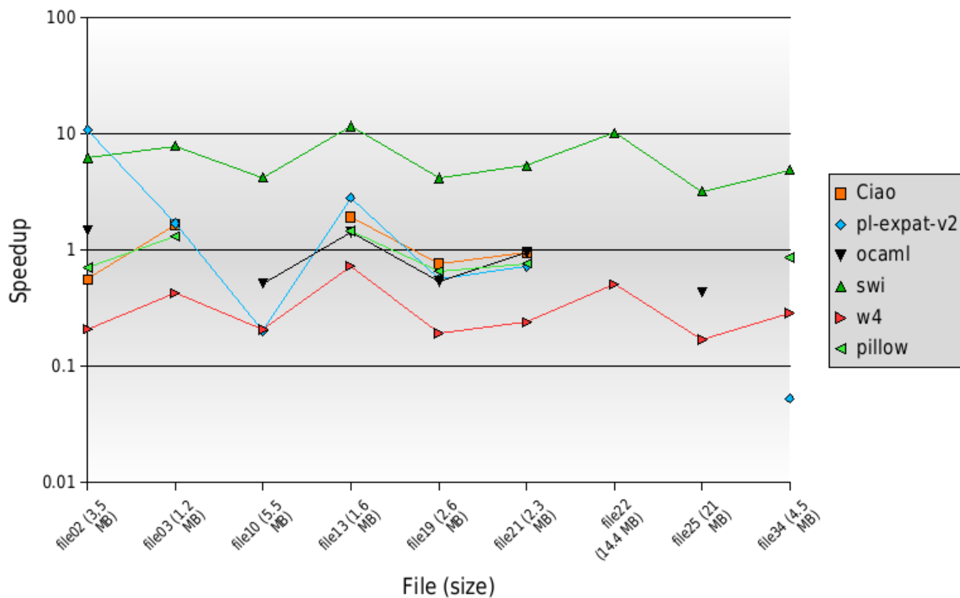


Figure 4.27: Speedup graph

### 4.5.2 Ontology representation

Next the benchmarks of the complete representation of the ontology are presented. For XPTO this includes the parse, semantic analysis, generation, compilation and loading of the units.

In addition to XPTO, in the tests were included other systems that provide similar capabilities: Thea and Pellet. These systems are further described in Section 1.3. Pellet implements a species verification when parsing the ontology, the times were measured with this feature disabled. Thea represents the ontology as predicates stored in the Prolog Knowledge Base. The representation of the ontology adopted in XPTO is described in Section 4.2.1.

Table 4.5 contains the times measured for all the systems, these times were measured using the same method as described in Section 4.5.1. The times of the XPTO system are further detailed in Section 4.5.3.

The values present in the *performance gain* table (Table 4.6) allow to compare the systems in terms of time of ontology representation. We can state that Pellet is the fastest of the benchmarked systems and that the XPTO is, on average, 97.5% times slower than the Pellet system. The XPTO system is further timed in Section 4.5.3 where explanations for the slowdown are given.

Table 4.5: Time (in seconds) of representing the ontologies

| File (Size) | Thea | XPTO | Pellet |
|---|---|---|---|
| file35 (2.3 MB) | 21.22 | 206.33 | 4.39 |
| file36 (1.2 MB) | 6.96 | 98.78 | 2.61 |
| file37 (2.2 MB) | 105.15 | 204.91 | 5.57 |
| file38 (1.2 MB) | 4.66 | 96.4 | 2.5 |

Table 4.6: Performance gain of representing the ontologies

| File (Size) | Thea | XPTO | Pellet |
|---|---|---|---|
| file35 (2.3 MB) | -79.33% | -97.87% | 0.00% |
| file36 (1.2 MB) | -62.45% | -97.35% | 0.00% |
| file37 (2.2 MB) | -94.71% | -97.28% | 0.00% |
| file38 (1.2 MB) | -46.29% | -97.41% | 0.00% |

### 4.5.3 XPTO time analysis

In this section is analyzed the time it takes for XPTO to parse each file. These times are measured using the `statistics/2` predicate of GNU Prolog, using the `real_time` statistics key.[5]

The times are presented in Table 4.7 and the parts of the system that were measured are:

**parse:** This represents the time it takes for the ontology file to be parsed using Expat (as explained in Section 4.1, page 28).

**build:** Is the time to build the dictionary.

**print:** Corresponds to the time used in generating the ontology representation files.

**compile:** Is the time it takes to compile all the generated files.

**load:** Is the time of dynamically loading the ontology into the running instance of the program.

As presented in the average times of each step on Table 4.7, it is possible to realize that most of the time used to integrate the ontology into the system is spent in external processes: compiling and loading the ontology takes over 90% of the process time.

[5]Further information about this predicate can be found in the GNU Prolog manual available at `http://www.gprolog.org/manual/gprolog.html#htoc232`

Table 4.7: Average time of each part of the representation time

| File (Size) | parse | build | print | compile | load |
|---|---|---|---|---|---|
| file35 (2.3 MB) | 0.01 | 0.03 | 0.02 | 0.79 | 0.15 |
| file36 (1.2 MB) | 0 | 0.04 | 0.02 | 0.86 | 0.07 |
| file37 (2.2 MB) | 0.01 | 0.04 | 0.01 | 0.8 | 0.14 |
| file38 (1.2 MB) | 0.01 | 0.03 | 0 | 0.89 | 0.07 |
| **Average** | **0.75%** | **3.50%** | **1.25%** | **83.50%** | **10.75%** |

This indicates that the compilation process should be done separately and build an executable with the representation of the ontology that can, at a later time, be loaded and queried.

## 4.6 Conclusion

This chapter presented the prototype system for representing and querying ontologies that was developed. The representation used for the ontology was described and also shown were the possibilities of querying the representation and some use cases of the implemented system.

Although the capabilities of the XPTO system are enough for a prototype status, some improvements must be performed to allow a more wide spread use:

- improve the parser to handle larger ontologies.
- enable loading several ontologies at the same time.
- improving the support for the semantics of the OWL language.

# Chapter 5

# SPARQL Query Engine

This chapter describes the Front End (FE) of the system described in Chapter 4. The FE is the component of the application dedicated to SPARQL query resolution: it allows for the possibility of querying the internal representation of the ontology (described in Section 4.2.1) using the SPARQL query language. A schema of the structure of the FE is presented in Figure 5.1.

The FE is split into 3 parts: the parser, the query resolution and the returning of the results as XML.

The SPARQL query is parsed using Flex [Pax07] and Bison [ED07] to produce a GNU Prolog/CX context representing the query that is then activated to calculate the output and display the resulting XML. The implemented SPARQL parser follows the specifications of the language defined in [PS06] and the results are returned in XML as specified in [BB06].

The query examples in this section are presented in the SPARQL specifications [PS06] or are examples that query the Wine ontology [W3C06].

This chapter briefly describes the SPARQL query language in Section 5.1, the representation and resolution of queries (Sections 5.2 and 5.3 respectively) and the XML output of the system (Section 5.4). Section 5.5 shows some examples where the developed system can be used.

## 5.1 Querying in SPARQL

SPARQL (SPARQL Protocol and RDF Query Language) is a Candidate Recommendation for a RDF query language [PS06] and is under continued development towards becoming the standard query language for the Semantic Web [FLB+06].

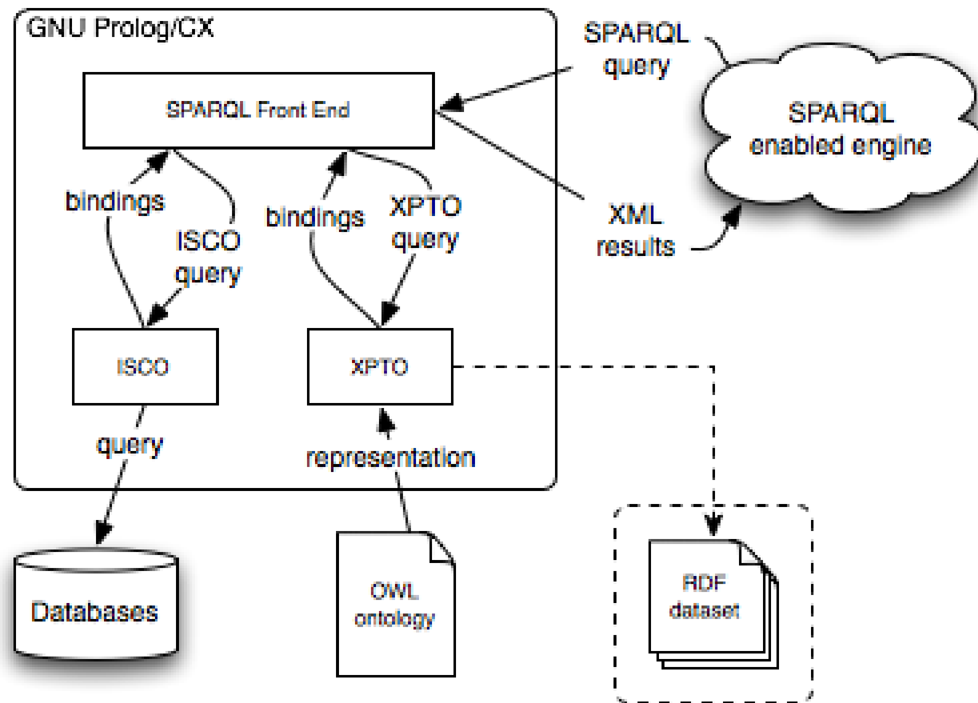


Figure 5.1: SPARQL query engine architecture

SPARQL has no inference engine inherent to the language, it merely specifies a syntax for the query and a means for returning the intended information.

At the time of the development of the system the current SPARQL specifications were: SPARQL Query Language for RDF - W3C Candidate Recommendation 6 April 2006 [PS06] and SPARQL Query Results XML Format - W3C Candidate Recommendation 6 April 2006 [BB06].

### 5.1.1 SPARQL query elements

The structure of a SPARQL query is illustrated in Table 5.1 (this representation is adapted from the one available in [Bec07]). For a more in depth explanation there are available the following documents: "SPARQL Query Language for RDF" [PS06], "SPARQL Protocol for RDF" [Cla06] and "SPARQL Query Results XML Format" [BB06].

Next are introduced some of the elements of SPARQL.

Table 5.1: SPARQL Query Language Structure (adapted)

| | |
|---|---|
| Prologue<br>(optional) | BASE <*iri*><br>PREFIX prefix: <*iri*> (repeatable) |
| Query Result forms<br>(required, choose 1) | SELECT (DISTINCT) {?*variable*, *}<br>DESCRIBE {?*variable*, <*iri*>, *}<br>CONSTRUCT graph pattern<br>ASK |
| Query Dataset Sources<br>(optional) | FROM <*iri*> (repeatable) |
| Add a named graph: | FROM NAMED <*iri*><br>(repeatable) |
| Graph Pattern<br>(optional, required for ASK) | WHERE graph pattern [ FILTER expression ] |
| Query Results Ordering<br>(optional) | ORDER BY ... |
| Query Results Selection<br>(optional) | LIMIT n, OFFSET m |

### RDF datasets

A SPARQL query can indicate the dataset that is to be used to perform the query matching. This is done using the `FROM` clause that specifies the IRI of the desired dataset. There is also available the `FROM NAMED` clause that can be used to add named graphs. These graphs can later be specifically queried using the `GRAPH` keyword.

### Variables

Variables in SPARQL are identified by the prefix `'?'` or `'$'` and can be present in any part of the graph pattern. Query variables have global scope; use of a given variable name anywhere in a query identifies the same variable. The prefix is not part of the variable name and represents the same variable with either prefix: in a SPARQL query, `$abc` and `?abc` are the same variable.

### Blank Nodes

A blank node is a node in an RDF graph that is not a URI reference or a literal, it corresponds to a unique node. A blank node can be present in the `subject` or `object` positions in an RDF triple and can be indicated using the label form: `_:a`, or by using `[]`. When written in the form `_:a`,

they represent a blank node with label `a`. Blank nodes that are used only once can be represented as `[]`, this will create a unique blank node and use it in the triple pattern. A more in depth analysis of blank nodes and their semantics can be found in [Par06].

The `[:p :v]` construct can also be used in triple patterns, this will create a blank node label which is used as the subject of all the contained pairs.

### Namespaces

Namespaces can be used to abbreviate IRIs using two different types: `BASE` and `PREFIX`. The same IRI can be represented using any of the following forms (shown in [PS06]):

**IRI:** IRIs are delimited by the atoms `'<'` and `'>'`, for instance:

```
<http://example.org/book/book1>
```

**Relative IRI:** The `BASE` keyword defines the IRI that is used to resolve relative IRIs:

```
BASE <http://example.org/book/>
<book1>
```

**Prefixed Name:** A *prefixed name* consists of two parts, the label and a local part, separated by a colon `':'`. The final IRI is obtained by the concatenation of the IRI associated with the prefix and the local part.

```
PREFIX book: <http://example.org/book/>
book:book1
```

### Basic Graph Pattern

A basic graph pattern is the core of the SPARQL language: it is responsible for connecting the query with the queried data. This way, basic graph patterns are represented using the triple form of RDF [KC04]: `subject`, `predicate` and `object`, with the possibility of any of them being a variable.

#### Group Graph Pattern

Group graph patterns are complex graph patterns that can be created by using simpler graph patterns such as basic graph patterns. A solution for a group graph pattern is any solution that is also a solution for each of the elements of the group graph pattern.

Using value constraints, optional graph patterns or alternative graph patterns are other ways of creating group graph patterns:

**Optional** The `OPTIONAL` statement indicates that the next triple pattern may not be bound in the solutions. If there is a graph pattern that matches the graph one or more pattern solutions will be generated, otherwise no additional bindings will be performed.

Optional patterns can occur inside any group graph pattern, including one that is itself an optional, thus forming a nested pattern. In this case, the outer optional graph pattern must match before any of the inner optional pattern can be matched.

**Union** The `UNION` operator provides the means of combining graph patterns. It allows solutions that match only one of the specified graph pattern, to be considered a solution of the group graph pattern.

**Value Constraints** The `FILTER` operator can be used to constrain the value of a variable based on an arithmetic expression, string contents or other operators and functions defined in [PS06].

### 5.1.2 Query forms and results

The result of a query is a sequence of solutions, with the variables instantiated by matching against the data in the dataset. A solution can consist of several bindings for the variables and the sequence in which they are shown can be modified using the solution modifiers listed in Section 5.1.3. By default, query patterns generate unordered solutions sequences.

After sequence modifiers are applied to the bindings, the SPARQL results format is determined based on the available query forms:

**select:** Returns the selected variables bound with the results.

**construct:** Returns an RDF graph with the structure indicated in the query and the variables instantiated.

**describe:** Returns the information known about the resources as an RDF graph.

**ask:** Returns a boolean indicating whether the specified query matches the data.

### SELECT query

There are two essential parts to a SPARQL select query: the `SELECT` clause and the `WHERE` clause. They allow the formulation a simple query in which the `SELECT` clause indicates the variables to be presented in the results and the `WHERE` clause consists of a graph pattern representing the conditions of the query. An example of such a query is shown in Figure 5.2.

```
1 SELECT ?title
2 WHERE
3 {
4  <http://example.org/book1>
5  <http://purl.org/dc/elements/1.1/title>
6  ?title .
7 }
```

Figure 5.2: SPARQL query example

## 5.1.3 Solution modifiers

The available solution modifiers are: `order by`, `distinct`, `offset` and `limit`. They are explained next:

**order by** The `order by` statement enables to define an order for the (otherwise unordered) solution sequence. This will order the results using the expressions present in the statement. It is also possible to define the direction by using the keywords `asc` for ascending order or `desc` for descending. By default the direction is ascending.

If this statement is not present in the query the order of the results is undefined and may even be different between equal queries.

Specifying the order of a sequence of solutions does not change the number of solutions of a query.

**distinct** Using the `distinct` keyword in the query will ensure that each solution in the sequence is unique, i.e., all the elements in the sequence are different.

**offset** will cause the specified number of solutions to be discarded from the beginning of the solution set. This enables ignoring part of the solutions that may have already been returned in a previous query.

Using `limit` and `offset` to select different subsets of the query solutions will not be useful unless the order is specified by using `order by` statement.

**limit** Using the `limit` statement will indicate that the query shall return at most the specified number of solutions. If the actual number of solutions to the query is greater than the specified limit, all the extra solutions are discarded.

### 5.1.4 Querying OWL ontologies using SPARQL

SPARQL is a query language for RDF and the semantics of using it to query OWL ontologies are not completely defined. To overcome this situation, a proposal for a subset of the SPARQL language with defined semantics for querying OWL DL is presented in [SP07]. Although not implemented in the XPTO system, SPARQL should be used to query RDF datasets. A possible implementation of this feature would be to enable XPTO to access RDF datasets as shown in Figure 5.1 (page 57).

The developed system is using SPARQL to query an ontology, allowing access to properties and resulting in individuals and property values in the same way as other available systems: Protégé [Knu07], Pellet [SPG+07] or Jena [Jen06].

## 5.2 Representation of a SPARQL query

The query representation process consists of a SPARQL parser that converts a query defined in the SPARQL syntax [PS06] into a GNU Prolog/CX context. This context represents the entire query and can then used to return the results. The execution of the generated context, triggered by a default message, that will bind the variables present in the query and show the results.

### 5.2.1 Element representation

The representation of query elements, such as SPARQL variables and resources, is presented next.

### Variables

The SPARQL variables are represented as Prolog variables. Thus, once the result is calculated, the query resolution system simply binds the corresponding variable to return the results.

There are some other structures needed to display the results: it is necessary to store the name of the variable in the SPARQL query in order to return it in the results. To achieve this, all the variables in the SPARQL query are stored in a list that will be the argument of the unit `vars/1`. The elements of this list are in the format `SparqlVariableName = PrologVariable`.

`SparqlVariableName` corresponds to the name of the variable in the SPARQL query and `PrologVariable` is the Prolog variable assigned to represent it. `PrologVariable` will start unbound and, as the context is resolved, will be instantiated with the solutions it may have.

In the case of a `select` query, the SPARQL variables that are to be shown in the results are stored as the argument of the unit `select/1`, using the same representation as unit `vars/1`.

SPARQL variables appear in the generated context for the query using the `PrologVariable` representation, enabling a simple access to the value of the variable or direct instantiation of an unbound variable. This representation can be seen in the GNU Prolog/CX context shown in Figure 5.4.

### Resources

Resources are represented using Prolog terms or atoms. If the resource is an absolute IRI (delimited by `'<'` and `'>'`) it is represented as an atom containing the entire IRI. For example the IRI

```
<http://example.org/book/book1>
```

is represented as

```
'http://example.org/book/book1'
```

If it corresponds to a prefixed name (a prefix label and a local part separated by a colon `':'`) it is represented as Prolog compound term of arity 2 with the functor ':'. The arguments of the term are the prefix name and the local part respectively. This representation is similar to the representation `ElementName` presented in Section 4.1. If the prefix name is empty the atom `''` will be used to represent it. The following prefixed IRI

```
PREFIX  : <http://example.org/book/>
:book1
```

is represented as

```
'':book1
```

This representation allows for the IRI to be resolved using the information stored in the unit `prefix` which contains the prefixes specified in the SPARQL query. This is further described in Section 5.3.1.

### 5.2.2 Query representation

A SPARQL query is represented as GNU Prolog/CX context whose structure is similar to the structure of the query. The elements of the query can be clearly identified in the representation: `select`, `where` as well as the *Modifiers* (if there are any present in the query).

```
SELECT
       ?flavor ?color
WHERE {
   ?t :hasFlavor    ?flavor  .
   ?t :hasColor      ?color  .
}
```

Figure 5.3: Query example (simple select)

```
[ where([set([
             triple(A,hasFlavor,B),
             triple(A,hasColor,C) ])
       ]),
 select([flavor=B,color=C]),
 vars([flavor=B,color=C,t=A]) ]
```

Figure 5.4: Generated context (partial) for the query in Figure 5.3

The example query presented in Figure 5.3 is a `select` query containing two basic graph patterns with a shared variable: `?t` and the context produced by the parser in shown in Figure 5.4. The order in which units may appear in the context is shown explained in Table 5.2. The units are further described in Section 5.3.1.

Table 5.2: Query context structure

| Unit/Arity | Description |
|---|---|
| `limit/1` | Optional |
| `offset/1` | Optional |
| `order/1` | Optional |
| `where/1` | Query conditions |
| `from/1` | Indicates RDF datasets |
| `select/1` or `ask/0` | Indicates the type of query |
| `prefix/1` | Prefixed Namespaces |
| `base/1` | Base Namespace |
| `vars/1` | Contains all the variables in the query |

A context is represented by a Prolog list containing unit names. The first element of the list will be the unit that first tries to evaluate the goal upon execution. The individuals and property values are gathered from the units in a higher position in the context. This way in the final positions of the list are found the units `select/1` (in the case of a select query) and `vars/1`. These units contain in their arguments a list of variables and will allow any unit in the context to access either all the variables in the context or the selected variables.

### Context examples

In addition to the partial context example presented in Figure 5.4, a complete context is presented in Figure 5.6 (page 67) and further explained.

Each of the elements present in the SPARQL query is represented in the generated context by one or more parametrized units. The already described `vars/1` and `select/1` units hold the all the variables present in the query and the variables that are to be returned, respectively.

Although not presented in Figure 5.3 the units `from`, `prefix` and `base` are always present in the generated context (even if absent in the query). If any of these keywords are not present in the query the corresponding unit will contain an empty list (this can be noted in the complete context presented in Figure 5.6).

A group of graph patterns, that appears in the SPARQL query enclosed by `'{'` and `'}'` is represented as the unit `set/1`, where the argument of the unit contains the representation (as units) of the enclosed graph patterns.

The unit that is the core of the query engine is `triple/3`. This unit will represent a simple graph pattern and the arguments of the unit are respectively the `subject`, `property` and `object` of the graph pattern. This

```
1 PREFIX foaf: <http://xmlns.com/foaf/0.1/>
2 PREFIX dc:   <http://purl.org/dc/elements/1.1/>
3 PREFIX xsd:   <http://www.w3.org/2001/XMLSchema#>
4 SELECT ?name
5  WHERE { ?x foaf:givenName  ?givenName .
6          OPTIONAL { ?x dc:date ?date } .
7          FILTER ( bound(?date) ) }
```

Figure 5.5: Query example

unit will be responsible for accessing the dataset in order to bind the variables that may appear in the arguments or test if the pattern has a solution.

Other presented units, such as the `optional/1` and `filter/1` units, are described in Section 5.3.1.

### 5.2.3 SPARQL parser

The SPARQL parser was implemented using Flex [Pax07] and Bison [ED07]. Flex and Bison are widely used in compiler construction and can be easily integrated with GNU Prolog through the use of `foreign` C declarations. The SPARQL parser could be implemented completely in Prolog using, for instance Definite Clause Grammars (DCGs). By using Flex and Bison, it was possible to use previously defined functions, namely to achieve the element representation described in Section 5.2.1.

In order to achieve the desired representation for the SPARQL query (described in Section 5.2.2) it was necessary to use additional structures in Bison (presented in Figure 5.7), mostly to represent the nested patterns. The query is represented in the structure *stack*. This structure contains an array of *levels* that will ultimately contain the whole query, the total size of the array and the number of elements already occupied.

Each *level* also contains an array of `PlTerm`[1] elements, in this case of the elements (that will be represented as units) of the depth level of the SPARQL query. It also contains the size (number of elements) and the index of the next available element.

Typically a new *level* is created when entering a grouped graph pattern (identified by the `'{'` character) and ended when leaving it (`'}'`). When ending a level a Prolog list with all the terms of that level is created and

[1]PlTerm is the C representation for a Prolog Term

```
[where([set([
            triple(A,foaf:givenName,B),
            optional([set([
                          triple(A,dc:date,C)
                          ])]),
            filter([bound(C)])
            ])]),
from([]),
select([name=D]),
prefix([foaf='http://xmlns.com/foaf/0.1/',
       dc='http://purl.org/dc/elements/1.1/',
       xsd='http://www.w3.org/2001/XMLSchema#']),
base([]),
vars([name=D,x=A,givenName=B,date=C])]
```

Figure 5.6: Context generated for the query in Figure 5.5

stored as the next element of its parent *level*.

The generated parser was tested against a set of the most common SPARQL queries and against SPARQL syntax examples present in [PS06]. Although there are cases not being handled by the resolution system (further described in Section 5.3) the parser itself is able to generate the correct context for the query.

## 5.3 SPARQL resolution system

The SPARQL query language is based on matching graph patterns. A basic graph pattern is the triple pattern, similar to an RDF triple, but with the possibility of a variable begin present in the subject, predicate or object positions.

> "A pattern solution can then be defined as follows: to match a basic graph pattern under simple entailment, it is possible to proceed by finding a mapping from blank nodes and variables in the basic graph pattern to terms in the graph being matched; a pattern solution is then a mapping restricted to just the variables, possibly with blank nodes renamed." in [PS06].

```
struct level {
  int size;
  int pos;
  PlTerm * array;
};

struct stack {
  int size;
  int pos;
  struct level ** array;
};
```

Figure 5.7: Auxiliary Parser structures

All the units that are present in the context generated by the SPARQL parser will answer to the goal `item/0` (or `item/1` in case of the unit returning a solution). Each unit will then perform the operation it implements, based on its arguments and on the result of the parent context. The query resolution is triggered by evaluating the goal `item/0` in the context returned by the mapping process.

The following guidelines were followed when implementing the units:

- each unit defines the interface predicates `item/0` and `item/1`. These are the predicates that trigger the *semantics* of the unit.
- Units that change the result set retrieve all the results from the super context (using the a predicate call similar to: `:^ item(I)`). These units then operate over this set of results and return one by one the results that are valid.

### 5.3.1 Unit description

In this section are presented some of the units that can be included in the context that represents the SPARQL query and are detailed the most important ones.

**Query form** These units indicate the type of query:

**ask/0** Indicates the specified query is an `ask` query.

**select/1** Indicates that the query is a `select` query. The unit argument contains a list of the variables that are to be returned in the query results.

**Solution Modifiers** The following units allow to restrict the results or change the result set:

**distinct/0** Eliminates the repetitions in the result set.

**filter/1** Selects the elements of the result set based on the restrictions provided.

**limit/1** Returns only the indicated number of elements in the results.

**offset/1** Eliminates the specified number of solutions from the beginning of the results set.

**optional/1** This unit is used to indicate that individuals that do not match the specified graph pattern are also to be included in the results.

**order/1** Orders the solution set according to the conditions provided.

**union/1** This unit is the representation of the SPARQL `UNION` operator that allows to combine several graph patterns.

**Other** These are auxiliary units or the implementation of other SPARQL operators:

**prefix/1** The argument of this unit is instantiated with a list containing the namespaces present in the query.

**set/1** This unit represents a group graph pattern in the query and allows the presence of nested patterns.

**triple/3** This is the unit that is responsible for binding the query variables. There are also available the `triple/4` and `triple/5` units.

**vars/1** Contains a list of all the variables present in the query.

**where/1** Contains the representation of the graph patterns to be used to select the individuals.

Next are detailed some of the most important units, each of these units redefine the goals `item/0` and `item/1`. The normal workflow of these predicates is to retrieve the results from the "super-context" using the operator "`:^`", perform the intended operation on the results and return them one by one using the Prolog backtrack mechanism.

**triple/3**

The core unit in the query resolution process is the `triple/3` unit, which is responsible for instantiating the variables in the query by accessing the data.

This unit can be redefined in order to access data available from different sources. The implementation of this unit to access the ontology representation described in Section 4.2.1 is shown in Figure 5.8. It generates one query to the XPTO system for each property that appears in the SPARQL query. The pattern in line 2 of Figure 5.4 (page 64) will generate the following query:

```
/> property(hasFlavor,F) :> item(I).
```

As explained in Section 4.3 (page 39) the argument of the `item/1` goal will be instantiated with the name of the individual. The arguments of the unit `property/2` are the name of the property being queried and the value of that property for the returned individual. Using the `property` unit to query the internal representation has the advantage of being able to perform the query using a Prolog variable in the position of the property name, thus enabling to return all the properties of the individual or querying the property name based on the property value.

```
1  :- unit(triple(S, P, O)).
2
3  item :-
4          /> property(P,O) :> item(S).
```

Figure 5.8: Unit `triple/3`

There are also available the units `triple/4` and `triple/5` that are the representation assigned to the `','` and `';'` SPARQL notations. The `';'` notation indicates that triple patterns have a common subject (it is only necessary to write it once) and `','` indicates that the patterns share both subject and predicate.[2] These units simply call the `triple/3` unit with the correct arguments.

These representations could be handled in the parser (generating two different `triple/3` units). The approach that was followed makes the representation more complex (introducing two new units) but enables for improvements to be done by redefining these units to perform different types of operations.

[2]Further information is available in `http://www.w3.org/TR/rdf-sparql-query/#predObjLists`

#### offset/1 and limit/1

These units change the number of results returned by the query. Both of the units contain an argument that must be an integer. The `offset/1` unit will discard the given number of solutions from the beginning of the solution set and the `limit/1` will cause the solution set to contain at most that number of solutions.

```
1 SELECT  ?name
2 WHERE   { ?x foaf:name ?name }
3 ORDER BY ?name
4 LIMIT   5
5 OFFSET  10
```

Figure 5.9: `limit` and `offset` query example

#### optional/1

The `optional` keyword in SPARQL defines that a graph pattern does not cause the query to fail if it has no bindings. The corresponding unit, `optional/1`, receives as input the representation of the graph pattern of the SPARQL query and resolves the context (by sending it the `item/1` goal). If the goal succeeds the variables present in the context are bound to its values and these values will be shown in the query results. On the other hand, if the `item/1` goal fails, the goal in the `optional/1` unit will succeed without instantiating any of the variables. Since unbound variables are not returned in the results these variables will not be shown in the XML output.

This unit also checks, in accordance to the `optional` keyword definition, if there is not a more specific solution to the query i.e., a solution with less unbound variables. Any given solution is only considered to be a valid solution if there is no other solution, with the same values for the instantiated variables, that contains less unbound variables. In order to perform this check, the unit, after retrieving all the solutions from the representation of the group graph pattern in its argument, removes the solutions that are not valid.

#### distinct/0

This unit, after gathering all the results from the context, removes the duplicates by checking if an element is present in the rest of the list of results.

The comparison predicate compares the values without instantiating any Prolog variables that may still be present in the results set. The `item/1` goal then instantiates its argument traversing the list using the `member/2` Prolog predicate.

#### filter/1

The `filter` keyword restricts the results of the query according to the expressions provided. It can be used to select the results of the query based the values of the variables, only returning those that correspond to a successful evaluation of the expression.

Currently this is implemented in the same way as the previously described units: collecting all the solutions and then selecting the valid results. A simple improvement that can be done is to restrict the results before retrieving them with the use of constraints.

Also, at this point, it is only possible to use numerical expressions to filter the results: the SPARQL builtin functions `STR`, `LANG`, `LANGMATCHES`, `DATATYPE`, `BOUND`, `sameTerm`, `isIRI`, `isURI`, `isBLANK`, `isLITERAL` and `REGEX` as well as function calls are not supported.

### 5.3.2 Unimplemented features

Besides the partially implemented `filter` operator, some SPARQL features are not currently implemented; these are now described:

**from** The `from` clause has no effect since the query resolution is done over a previously loaded ontology. This implies that it is not possible to specify an external ontology and run the query over that ontology.

**namespaces** As is done in the XPTO system, namespaces are currently being ignored in the SPARQL front end. All matching is done against the internal representation of the ontology that is considered the `BASE` ontology.

**describe** The `describe` statement is indicated as returning a unconstrained information about the node and is currently marked as "Feature at Risk" as stated in [PS06]:

> "The DESCRIBE feature of SPARQL is an intentionally unconstrained query feature. On the one hand, it has been the subject of a number of critical comments (...); on the

other hand, it is required by a number of interesting semi-structured query use cases (. . . ) the feature has been marked non-normative and at-risk (. . . )"

**construct** The `construct` statement returns an RDF graph with the structure specified in the query. It is possible to implement this feature by defining the output unit that shows the correctly formatted output (the other units should be used without any change).

## 5.4 Returning Query Results

As explained in the previous sections the structure of the XML output of the SPARQL query is decided by the inclusion in the context of the units `select/1` or `ask/0`. These unis are responsible for retrieving the query bindings from the context and building the XML that corresponds to the type of query.

As presented in [BB06], the XML format of the SPARQL query results is a XML document with a `sparql` element. This element always contains an element `head` as the first sub-element. The elements after this one are specific to the form of query.

The XML is generated using the PiLLoW library (described in Section 3.4, page 25).

### 5.4.1 Select query

For a `select` query, the element after the `head` element, is the `results` element. In the `select` query the `head` element contains a list of all the variable names present in the SPARQL query ordered in the order they appear in the query.

The second element (`results`) is a list of `result` elements, each containing bindings for a variable present in the query. The `results` element has two boolean attributes: ordered and distinct, that are always specified. They indicate, respectively, if the list of results is ordered and if the elements are all different. The value of these attributes is defined by the presence or absence of the modifiers `distinct` and `order by` in the SPARQL query. The resulting XML of the query in Figure 5.3 (page 64) is shown in Figure 5.10.

### 5.4.2 Ask query

For an `ask` query (that only returns a boolean), after the `head` element, there is a single element with the name `boolean` that indicates if the specified

```
<?xml version="1.0"?>
<sparql>
  <head>
    <variable name="flavor"></variable>
    <variable name="color"></variable>
  </head>
  <results ordered="false" distinct="false">
    <result>
      <binding name="flavour">Medium</binding>
      <binding name="color">White</binding>
    </result>
  </results>
</sparql>
```

Figure 5.10: XML output of the query example in Figure 5.3

query is `true` or `false` (as can be seen in Figure 5.11).

```
<?xml version="1.0"?>
<sparql>
  <head></head>
  <boolean>true</boolean>
</sparql>
```

Figure 5.11: XML output for an `ASK` query

## 5.5 Examples

Here are presented two examples of use for the developed system (XPTO and SPARQL front-end). One example uses only the developed SPARQL parser and resolution system to enable querying relational databases in SPARQL. The other consists of a implementation of a SPARQL Web Service allowing the system to be queried over the Web.

### 5.5.1 Using SPARQL to query a relational database

By using the ISCO [AN06] framework it is possible to enable using SPARQL to query a relational database. In this example, the database that was used contains data relative to an Academic Services application: Universidade de Evora's Integrated Information System (SIIUE) [GQA03]. An example of a SPARQL query that can be performed is shown in Figure 5.13 and the queried relation is represented, using the ISCO syntax, in Figure 5.12.

In order to be able to query fields with the same name from different relations, it is necessary to define that each field name is prefixed by the name of the relation and an '_' forming: `RelationName_FieldName`. This way, the `:student_number` query in line 2 of Figure 5.13 represents the field `number` of table `student`. The name of the individual, that will be mapped to each tuple in the relation, is the Postgresql internal `OID`[3] of the tuple.

The query presented in Figure 5.13 selects the students (represented by the OID of the tuple), the number and institution of the student, for the students whose number is between 300 and 500. The context that is generated by the SPARQL parser is shown in Figure 5.14 and the SQL queries generated by the ISCO framework to access the database is shown in Figure 5.15.

```
1 mutable class student.
2   id: individual.id. unique.
3   number: int. unique.
4   institution: institution.id.
```

Figure 5.12: ISCO definition of the relation `student`

This feature is implemented as an example and there are several improvements to be made in order to make it efficient. Some are, for instance:

- allow to query more than one relation field. As the query is being translated to the ISCO language it is performing one query to the database for each property present in the SPARQL query. This could be improved by detecting patterns in the query and rewriting it to minimize the number of queries to the database.

[3] The OID is a unique number across the entire installation automatically assigned to a row and that identifies it. PostgreSQL uses OIDs to link its internal system tables together. Further information can be found in `http://www.postgresql.org/docs/8.2/static/datatype-oid.html`

```
SELECT ?number ?inst WHERE {
  ?student :student_number ?number .
  ?student :student_institution ?inst .
  FILTER ( ?c > 300 && ?c < 500  )
}
```

Figure 5.13: using SPARQL to query a relational database

```
[where([set([
             triple(A,'':student_number,B),
             triple(A,'':student_institution,C),
             filter([and(bigger(D,300),smaller(D,500))])
             ])]),
from([]),
select([number=B,inst=C]),
prefix([]),
base([]),
vars([number=B,inst=C,student=A,c=D])]
```

Figure 5.14: Generated context for the query in Figure 5.13

- enable filtering the elements before they are retrieved from the relation. Currently all the elements are gathered from the relation being queried and filtered later in the `filter` statement.

### 5.5.2 SPARQL Web service

As another example of the developed system is a Web interface that was built in order to allow answering of SPARQL queries over the Web. This consists of a simple user interface in which users can specify the queries and retrieve the results. It is possible to use XML Transformation languages like XSLT [Cla99] to change the presentation of the results in order to be displayed in a user-friendly manner.

There is also available a version in which the query is specified as part of the URL and the results are then returned to the browser or agent. This form of input is mostly aimed for automatic use by a SPARQL agent.

The SPARQL web service example may work in two different ways:

```
select o.oid, 'student' as instanceOf, o."number"
from "student" o;

select o.oid, 'student' as instanceOf, o."institution"
from "student" o where o."oid"=19918;
```

Figure 5.15: SQL queries generated for the context in Figure 5.14

- the XPTO system has to load the ontology for each query that is performed. This would allow, in each query, to specify the dataset to be used to perform the matching.
- the dataset over which the query will be performed is already integrated with the system but, in this case, there is only the possibility of querying the available dataset.

The example implements the second method: the ontology is integrated with the system and the SPARQL query will be performed over that ontology. This is done because, as presented in Section 4.5.2, loading the ontology is a slow process and it would cause the overall time (loading and querying the ontology) to be unsustainable.

## 5.6 Conclusion

This chapter has described the approach taken to add to the XPTO system the capabilities of being queried using the SPARQL query language. Each query is mapped into a GNU Prolog/CX context and executing it obtains the results of the query. These results are returned using the XML structure defined by the SPARQL protocol.

# Chapter 6

# Conclusion

The main objective of this work was to introduce a framework for accessing web ontologies using Contextual Logic Programming (CxLP) that also supports integration of information from other data sources.

The most important integration is with relational databases using the ISCO framework, this enables the database to be queried either inside the GNU Prolog/CX environment or by using the SPARQL query language.

The developed system, XPTO, is able to represent an ontology described in OWL DL and enables querying the ontology based on the generated representation. The representation consists of CxLP units and querying can be performed by building a context that includes the units of the representation.

The loaded ontology is represented using several units: the unit named `ontologies` is used to represent data about the ontology such as the properties and classes it contains. Another unit, the `individuals` unit, is used to represent the individuals of the ontology, the individual relations and the class memberships. There is also one unit for each class and property present in the ontology. Each unit will contain information about the element it represents.

With this representation, querying can be performed by using a specific operator that was introduced for this action: `'/>'`. This operator can optionally be preceded by the unit that represents a class in order to query only the elements of that specific class. It can also be followed by the unit of a property to query the value of the individual for that property, or several other defined units to perform actions such as querying the property name based on the property value or indicating *optional* parts of the query.

A front-end capable of answering queries expressed using SPARQL was also developed, meant to act as a web service, it can access data available in several repositories, such as relational databases or the representation of

the ontology. This front-end acts as a translator: mapping SPARQL queries to a representation of the query that is based on CxLP units. In this representation each operator and each part of the SPARQL query corresponds to a unit occurring in the context and the complete query is represented by a context that combines the available units. The triggering of the context resolution performs the desired effect either accessing the ontology representation for retrieving the results or performing the several SPARQL operations for manipulating the answer set.

The presented representation of the ontology was not the first approach, having suffered two major changes:

- The first approach was to represent the individuals and their properties as instantiations of the classes: they would be represented as a compound term containing the name of the class as functor and list the name of the individual and all the properties as arguments in a predefined order. The individuals were stored in the class they belonged to.

  This representation was later abandoned due to the difference in the number of properties each individual could contain: each class unit would have one argument for each property defined in the ontology.

- Later, a representation in which each individual was represented in a different unit was tested. This representation although being the most modular and representative of the original ontology structure was also replaced by the described structure, in which all the individuals of the ontology are stored in the same unit. This allows to take advantage of some Prolog predicate optimizations such as indexing (as described in [AK91]).

Some presented benchmarks show the developed system is slow in representing the ontologies when compared to other systems. Also presented where the reasons for this slowdown: the compilation and loading of the generated units. This indicates that the XPTO system should be used to generate an executable with the representation of the ontology instead of loading the units on the fly. This executable can be loaded at a later time to perform queries on the ontology.

# Future Work

Further improvements can be performed on the developed system, both in the ontology representation layer and on the SPARQL resolution. On the representation side one can point out:

**Allow multiple ontologies to be loaded:** Currently it is only possible to work with one ontology at a time. Allowing for an arbitrary number of ontologies to be loaded and thus enable querying them, is considered to be an essential part of improving the system. This will probably require a change in the representation of the ontology, at least in the naming of the units that represent the classes and properties of the ontology.

**Semantics of OWL:** The several OWL constructors are only currently being stored in the unit that represents the element they refer. It is also vital the development of the system into a more elaborate reasoning system, that the semantics of these constructors be taken into account and correctly mapped in the representation of the ontology. To achieve this, there is a lot of space for improvement in the *micro-representation* of the ontology, i.e., the predicates that are present in each unit of the representation, or by developing rules that implement the semantics.

**Strengthen the integration with ISCO:** Although this was one of the main objective and motivation for this work this integration is still very small. To further integrate the system with ISCO, it should be possible to describe an ontology in a similar form to the ISCO description of a database (as presented in [AN06]) and to develop a uniform and desirably indistinct way to query both an ontology and a relational database.

In the SPARQL query answering component the more relevant developments are:

**Complete the SPARQL support:** Currently not all of the SPARQL constructors are implemented. Although not necessary to achieve the desired prototype status of the system, a full support for the SPARQL specifications and protocol is necessary for the deployment of the system.

**Adopt the latest SPARQL specifications:** The SPARQL system was developed against the specifications of 6 April 2006 in which SPARQL was considered W3C Candidate Recommendation. There have been further

developments to the SPARQL language since then and it is necessary to update the implementation.

Throughout the development of this work some choices had to be made: from choices of representation to language choices. In terms of representation, after some evolution, the achieved representation is considered to be efficient and straightforward. In the field of languages, the most difficult choice was SPARQL since, although there are no valid alternatives, there are also undefined issues when using SPARQL to query OWL ontologies.

This concludes the description of the developed system. XPTO allows to represent, query OWL ontologies and answer SPARQL queries using GNU Prolog/CX.

# Bibliography


[AD03] Salvador Abreu and Daniel Diaz. Objective: in Minimum Context. In Catuscia Palamidessi, editor, *Logic Programming, 19th International Conference, ICLP 2003, Mumbai, India, December 9-13, 2003, Proceedings*, volume 2916 of *Lecture Notes in Computer Science*, pages 128–147. Springer-Verlag, 2003. ISBN 3-540-20642-6.

[AK91] Hassan Aït-Kaci. *Warren's abstract machine: a tutorial reconstruction.* MIT Press, Cambridge, MA, USA, 1991.

[AN06] Salvador Abreu and Vítor Nogueira. Using a Logic Programming Language with Persistence and Contexts. In Masanobu Umeda and Armin Wolf, editors, *Declarative Programming for Knowledge Management*, volume 4369 of *LNCS*, Fukuoka, Japan, 2006. Springer.

[AvH04] Grigoris Antoniou and Frank van Harmelen. *A Semantic Web Primer.* The MIT Press, 2004.

[BB06] D. Beckett and J. Broekstra. SPARQL Query Results XML Format. W3C recommendation, W3C, April 2006. Available at: `http://www.w3.org/TR/2006/CR-rdf-sparql-XMLres-20060406/`.

[BBF+06] Pedro Barahona, François Bry, Enrico Franconi, Nicola Henze, and Ulrike Sattler, editors. *Reasoning Web, Second International Summer School 2006, Lisbon, Portugal, September 4-8, 2006, Tutorial Lectures*, volume 4126 of *Lecture Notes in Computer Science.* Springer, 2006.

[BBFS05] James Bailey, François Bry, Tim Furche, and Sebastian Schaffert. Web and semantic web query languages: A survey. In Norbert Eisinger and Jan Maluszynski, editors, *Reasoning Web*, volume 3564 of *Lecture Notes in Computer Science*, pages 35–133. Springer, 2005.

[Bec07] Dave Beckett. SPARQL RDF Query Language Reference. Available at: http://www.dajobe.org/2005/04-sparql/SPARQLreference-1.8.pdf, 15 July 2007.

[BG04] D. Brickley and R. V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. W3C Recommendation, RDF Core Working Group, World Wide Web Consortium, 2004.

[BHS05] Franz Baader, Ian Horrocks, and Ulrike Sattler. Description logics as ontology languages for the semantic web. In Dieter Hutter and Werner Stephan, editors, *Mechanizing Mathematical Reasoning*, volume 2605 of *Lecture Notes in Computer Science*, pages 228–248. Springer, 2005.

[BK07] Harold Boley and Michael Kifer. RIF Core Design - W3C Working Draft 30 March 2007. Technical report, W3C, 2007.

[BLF99] Tim Berners-Lee and Mark Fischetti. *Weaving the Web : The Original Design and Ultimate Destiny of the World Wide Web by its Inventor*. Harper San Francisco, September 1999.

[BLHL01] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284(5), 2001.

[BPSM+06] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible Markup Language (XML) 1.0 (fourth edition). Technical report, W3C, 2006.

[CDA+06] Isabel F. Cruz, Stefan Decker, Dean Allemang, Chris Preist, Daniel Schwabe, Peter Mika, Michael Uschold, and Lora Aroyo, editors. *The Semantic Web - ISWC 2006, 5th International Semantic Web Conference, ISWC 2006, Athens, GA, USA, November 5-9, 2006, Proceedings*, volume 4273 of *Lecture Notes in Computer Science*. Springer, 2006.

[CH07] Daniel Cabeza and Manuel Hermenegildo. The PiLLoW Web Programming Library. Available at: http://www.clip.dia.fi.upm.es/Software/pillow/, 12 June 2007.

[CHV96] D. Cabeza, M. Hermenegildo, and S. Varma. The PiLLoW/CIAO Library for Internet/WWW Programming using

Computational Logic Systems. In P. Tarau, A. Davison, K. De-Bosschere, and M. Hermenegildo, editors, *Proc. 1st Workshop on Logic Programming Tools for INTERNET Applications*, page (Electronic proceedings), 1996.

[Cla99] J. Clark. XSL transformations (XSLT) version 1.0 W3C recommendation 16 november 1999. Technical report, W3C - World Wide Web Consortium, 1999.

[Cla06] Kendall Grant Clark. SPARQL Protocol for RDF. Technical report, W3C, 6 April 2006.

[Coo06] Clark Cooper. The Expat XML Parser Homepage. `http://expat.sourceforge.net/`, 27 November 2006.

[CR04] J. J. Carroll and J. De Roo. OWL web ontology language test cases. W3C recommendation, W3C, 2004. Available at http://www.w3.org/TR/2004/REC-owl-test-20040210/.

[Dam07] Carlos Viegas Damásio. W4 xml parser. http://centria.di.fct.unl.pt/˜cd/projectos/w4/xmlparser/index.htm, 20 February 2007.

[DAR07] DARPA. DAML. `http://www.daml.org/`, 3 February 2007.

[DFvH03] J. Davies, D. Fensel, and F. van Harmelen, editors. *Towards the Semantic Web: Ontology-Driven Knowledge Management*. John Wiley & Sons, 2003.

[DSB+04] M. Dean, G. Schreiber, S. Bechhofer, Frank van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. OWL web ontology language reference. W3C recommendation, W3C, Feb 2004. http://www.w3.org/TR/owl-ref/.

[DSW06] John Davies, Rudi Studer, and Paul Warren, editors. *Semantic Web Technologies - trends and research in ontology-based systems*. John Wiley & Sons, 2006.

[ED07] Paul Eggert and Akim Demaille. Bison - GNU parser generator. Available at: `http://www.gnu.org/software/bison/`, 21 February 2007.

[EK76] M. H. Van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *J. ACM*, 23(4):733–742, 1976.

[FHH04] Richard Fikes, Patrick J. Hayes, and Ian Horrocks. OWL-QL - a language for deductive query answering on the Semantic Web. *J. Web Sem.*, 2(1):19–29, 2004.

[FLA07] Cláudio Fernandes, Nuno Lopes, and Salvador Abreu. On querying ontologies with contextual logic programming. In Golbreich et al. [GKP07].

[FLB+06] Tim Furche, Benedikt Linse, François Bry, Dimitris Plexousakis, and Georg Gottlob. Rdf querying: Language constructs and evaluation methods compared. In Barahona et al. [BBF+06], pages 1–52.

[GH99] Daniel Cabeza Gras and Manuel V. Hermenegildo. The ciao module system: A new module system for prolog. *Electr. Notes Theor. Comput. Sci.*, 30(3), 1999.

[GH01] Daniel Cabeza Gras and Manuel V. Hermenegildo. Distributed www programming using (ciao-)prolog and the pillow library. *TPLP*, 1(3):251–282, 2001.

[GKP07] Christine Golbreich, Aditya Kalyanpur, and Bijan Parsia, editors. *OWL: Experiences and Directions 2007*, volume 258 of *CEUR Workshop Proceedings ISSN 1613-0073*, June 2007.

[GQA03] Joaquim Godinho, Luis Quintano, and Salvador Abreu. Universidade de Évora's Integrated Information System: An Application. In Hans Dijkman, Petra Smulders, Bas Cordewener, and Kurt de Belder, editors, *The 9th International Conference of European University Information Systems*, pages 469–473. Universiteit van Amsterdam, July 2003. ISBN 90-9017079-0.

[Gru93] Thomas R. Gruber. A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisitation*, 5(2):199–220, 1993.

[HBEV04] Peter Haase, Jeen Broekstra, Andreas Eberhart, and Raphael Volz. A comparison of RDF query languages. In *Proceedings of the Third International Semantic Web Conference, Hiroshima, Japan, 2004.*, NOV 2004.

[Hef04] Jeff Heflin. OWL Web Ontology Language Use Cases and Requirements. W3C Recommendation, World Wide Web Consortium, 2004.

[HM01] V. Haarslev and R. Möller. Description of the racer system and its applications. In *Proceedings International Workshop on Description Logics (DL-2001), Stanford, USA, 1.-3. August*, pages 131–141, 2001.

[ISO02] ISO/IEC. ISO/IEC 13250 Topic Maps. Technical report, ISO/IEC, 2002.

[Jen06] Jena. A Semantic Web Framework for Java. `http://jena.sourceforge.net/`, 30 November 2006.

[KC04] G. Klyne and J. Carroll. Resource description framework (RDF): Concepts and abstract syntax. W3C recommendation, W3C, 2004. Available at http://www.w3.org/TR/rdf-concepts/.

[KK01] José Kahan and Marja-Ritta Koivunen. Annotea: an open rdf infrastructure for shared web annotations. In *WWW '01: Proceedings of the 10th international conference on World Wide Web*, pages 623–632, New York, NY, USA, 2001. ACM Press.

[KM02] Marja-Riitta Koivunen and Eric Miller. W3C Semantic Web Activity. In E. Hyvonen, editor, *Semantic Web Kick-off in Finland*, pages 27–44, 2002.

[KMR04] Holger Knublauch, Mark A. Musen, and Alan L. Rector. Editing description logic ontologies with the protégé owl plugin. In Volker Haarslev and Ralf Möller, editors, *Description Logics*, volume 104 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2004.

[Knu07] Holger Knublauch. Protégé: Using sparql in Protégé-owl. `http://protege.stanford.edu/doc/sparql/`, 14 October 2007.

[Koi07] Marja-Riitta Koivunen. Annotea project. Available at: `http://www.w3.org/2001/Annotea/`, 12 June 2007.

[Kow74] Robert Kowalski. Predicate logic as programming language. In *IFIP Congress*, pages 569–574, 1974. Reprinted in Computers

for Artificial Intelligence Applications, (eds. Wah, B. and Li, G.-J.), IEEE Computer Society Press, Los Angeles, 1986, pp. 68–73.

[LA03] Ora Lassila and Mark Adler. Semantic gadgets: Ubiquitous computing meets the semantic web. In Dieter Fensel, James A. Hendler, Henry Lieberman, and Wolfgang Wahlster, editors, *Spinning the Semantic Web*, pages 363–376. MIT Press, 2003.

[LD01] Martin S. Lacher and Stefan Decker. RDF, Topic Maps, and the Semantic Web. *Markup Lang.*, 3(3):313–331, 2001.

[LFA07] Nuno Lopes, Cláudio Fernandes, and Salvador Abreu. Contextual logic programming for ontology representation and querying. In Axel Polleres, David Pearce, Stijn Heymans, and Edna Ruckhaus, editors, *2nd International Workshop on Applications of Logic Programming to the Web, Semantic Web and Semantic Web Services*, September 2007.

[MHRS06] Boris Motik, Ian Horrocks, Riccardo Rosati, and Ulrike Sattler. Can owl and logic programming live together happily ever after? In Cruz et al. [CDA+06], pages 501–514.

[MM04] Frank Manola and Eric Miller. RDF Primer. W3C Recommendation, World Wide Web Consortium, February 2004.

[MP89] Luís Monteiro and António Porto. Contextual logic programming. In *ICLP*, pages 284–299, 1989.

[MP93] Luís Monteiro and António Porto. A language for contextual logic programming. In *Logic programming languages: constraints, functions, and objects*, pages 115–147, Cambridge, MA, USA, 1993. MIT Press.

[MSR02] Libby Miller, Andy Seaborne, and Alberto Reggiori. Three implementations of squishql, a simple rdf query language. In Ian Horrocks and James A. Hendler, editors, *International Semantic Web Conference*, volume 2342 of *Lecture Notes in Computer Science*, pages 423–435. Springer, 2002.

[MvH04] Deborah L. McGuinness and Frank van Harmelen. OWL Web Ontology Language Overview. W3C Recommendation, World Wide Web Consortium, February 2004.

[NAD04] Vitor Beires Nogueira, Salvador Abreu, and Gabriel David. Towards temporal reasoning in constraint contextual logic programming. In *Proceedings of the 3rd International Workshop on Multiparadigm Constraint Programming Languages MultiCPL'04 associated to ICLP'04*, Saint–Malo, France, September 2004.

[PAG06] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and Complexity of SPARQL. In Cruz et al. [CDA+06], pages 30–43.

[Par06] Bijan Parsia. Querying the Web with SPARQL. In Barahona et al. [BBF+06], pages 53–67.

[Pas04] Thomas B. Passin. *Explorer's Guide to the Semantic Web*. Manning, Greenwich, 2004.

[Pax07] Vern Paxson. flex: The Fast Lexical Analyzer Manual. `http://www.gnu.org/software/flex/manual/html_mono/flex.html`, 21 February 2007.

[Pro06] Protégé. Free, open source ontology editor and knowledge-based framework. `http://protege.stanford.edu/`, 30 November 2006.

[PS06] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. Technical report, W3C, 2006. Available at: `http://www.w3.org/TR/2006/CR-rdf-sparql-query-20060406/`.

[Rém00] Didier Rémy. Using, understanding, and unraveling the ocaml language. from practice to theory and vice versa. In Gilles Barthe, Peter Dybjer, Luis Pinto, and João Saraiva, editors, *APPSEM*, volume 2395 of *Lecture Notes in Computer Science*, pages 413–536. Springer, 2000.

[SBLH06] Nigel Shadbolt, Tim Berners-Lee, and Wendy Hall. The Semantic Web Revisited. *IEEE Intelligent Systems*, 21(3):96–101, 2006.

[Sea04] Andy Seaborne. RDQL - A Query Language for RDF (Member Submission). Technical report, W3C, January 2004.

[Sof07] Software Systems Institute. Racer Manager. `http://racerproject.sourceforge.net/`, 19 February 2007.

[SP07] Evren Sirin and Bijan Parsia. Sparql-dl: Sparql query for owl-dl. In Golbreich et al. [GKP07].

[SPG+07] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical owl-dl reasoner. *J. Web Sem.*, 5(2):51–53, 2007.

[SS86] Leon Sterling and Ehud Y. Shapiro. *The Art of Prolog - Advanced Programming Techniques.* MIT Press, 1986.

[Sto07] Gerd Stolpmann. The xml parser for o'caml. http://www.ocaml-programming.de/programming/pxp.html, 27 March 2007.

[SWM04] Michael K. Smith, Chris Welty, and Deborah L. McGuinness. OWL Web Ontology Language Guide. W3C Recommendation, World Wide Web Consortium, February 2004.

[Top01] TopicMaps.org. XML Topic Maps (XTM) 1.0. Technical report, TopicMaps.org, 2001.

[Van06] Vangelis Vassiliadis. Thea OWL Parser for Prolog. `http://www.semanticweb.gr/TheaOWLParser/`, 12 October 2006.

[Van07] Vangelis Vassiliadis. *Thea A Web Ontology Language - OWL Parser for [SWI] Prolog.* http://www.semanticweb.gr/downloads/Thea%20OWL%20Parser.doc, 19 January 2007.

[Vat07] Irène Vatton. Amaya home page. `http://www.w3.org/Amaya/`, 12 June 2007.

[Vei06] Daniel Veillard. Libxml - the XML C parser and toolkit of Gnome. `http://xmlsoft.org/`, 27 November 2006.

[W3C06] W3C. Wine Ontology. `http://www.w3.org/TR/owl-guide/wine.rdf`, 22 July 2006.

[Wie03] Jan Wielemaker. An overview of the swi-prolog programming environment. In Frédéric Mesnard and Alexander Serebrenik, editors, *WLPE*, volume CW371 of *Report*, pages 1–16. Katholieke Universiteit Leuven, Department of Computer Science, Celestijnenlaan 200A, B-3001 Heverlee (Belgium), 2003.