Universidade de Évora

Master Degree in Computer Science
Mestrado em Engenharia Informática

X.P.T.O. - A system for representing and querying Semantic Web ontologies.

Cláudio Francisco Fernandes
<cff@di.uevora.pt>

Supervisor: *Salvador Abreu*

Évora, October 2007

Abstract

The Semantic Web represents an evolutionary step for the Internet where data is modeled to be semantically adequate also for machine agents. For this meta-info to be useful, systems must be built which can process it in order to infer knowledge.

We aim for the creation of a computational system, from a contextual logic programming point of view, that can process information from different sources in different formats.

Throughout this thesis we describe a prototype with two components which represent the main contributions of the work described herein:

- A core that is capable of representing Web ontologies;
- A back-end capable of mapping GNU Prolog/CX to SPARQL queries;

The core system acts like a computational hub for knowledge modeled by OWL ontologies that enables querying that representation. The back-end provides functions for communicating with SPARQL agents so that the reasoning of the internal knowledge base can be merged with external ontologies.

Resumo

X.P.T.O. - Um sistema de informação para representar e interrogar ontologias *Web*.

O termo *Semantic Web* representa um passo evolutivo para a Internet onde os dados são modelados de forma a serem semanticamente adequados também para agentes informáticos.

A necessidade de sistemas informáticos que consigam processar estes dados de forma a inferir conhecimento motivou o nosso objectivo: criar um sistema que consiga processar informação, de um ponto de vista da programação em lógica contextual, vindo de diferentes fontes em variados formatos.

Esta tese descreve um protótipo com dois componentes que representam a principal contribuição do trabalho efectuado:

- Um sistema capaz de representar ontologias Web;
- Um componente que traduz interrogações GNU Prolog/CX para SPARQL;

O sistema actua como um *hub* computacional para conhecimento descrito por ontologias OWL e permite fazer interrogações a essa base de conhecimento. O componente adicional permite fazer interrogações a agentes SPARQL externos, o que possibilita juntar o conhecimento vindo de fontes externas com o representado na base de conhecimento interno.

Acknowledgments

I would like to thank my mom, dad and brother. They are my heroes and my idols. Without their love and support none of this would be possible.

Thank you Rita for always believing in me. Without your love and support this would have been much harder.

Also I would like to thank my dear friends for their support, for reviewing my work and, most importantly, for making me a happy person. Thanks all of you guys.

Last but not least, thank you Nuno Lopes, for it has been a pleasure working with you. Thank you Salvador Abreu for the guidance provided and for giving me the chance to work and learn from you.

Contents

List of Figures

List of Tables

Definitions and Acronyms

XPTO : XPTO Prolog Translation for Ontologies

Arity : The number of arguments of a predicate or unit

CxLP : Contextual Logic Programming

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

Predicate : A relationship or statement of existence in a Logic Program

Functor : The name of a logic programming predicate

Unit : The CxLP representation of a set of Prolog predicates

HTML : HyperText Markup Language

OOP : Object Oriented Programming

Ontology : A data model that represents a set of concepts within a particular domain and the relationships between those concepts

OWL : Web Ontology Language

RDF : Resource Description Framework

RDFS : Resource Description Framework Schema

RQL : RDF Query Language

SPARQL : Simple Protocol and RDF Query Language

XML : eXtensible Markup Language

PiLLoW : Programming in (Constraint) Logic Languages on the Web. A library for handling WWW documents (HTML, XML) from Prolog

DAWG : Data Access Working Group

KB : Knowledge Base

FAQ : Frequently Asked Questions

W3C : World Wide Web Consortium

AJAX : Asynchronous JavaScript and XML

DOM : Document Object Model

MIME : Multipurpose Internet Mail Extensions

URI : Uniform Resource Identifier

IRI : International Resource Identifier

URL : Uniform Resource Locator

SOAP : Simple Object Access Protocol

WSDL : Web Services Description Language

XAK : XML Army Knife

SQL : Structured Query Language

ISCO : Information System COnstruction, a framework to access relational databases from Prolog

Chapter 1

Introduction

The work presented in this thesis is the result of a two years masters research that unifies the Semantic Web and Logic Programming topics in order to deliver the implementation of XPTO [1] and a SPARQL (Simple Protocol and RDF Query Language) back-end. XPTO is a contextual logic programming system capable of acting as a computational hub of information focused mainly in Web ontologies. The associated back-end is aimed to act as a communication socket with outside Semantic Web agents for querying OWL (Web Ontology Language) Web ontologies using the SPARQL query language.

Part of the work described herein has been previously presented. An initial description was shown in [LFA07]. Use cases and examples were presented in [FLA07].

1.1 Background

The Semantic Web. This phrase represents a concept that can bring great excitement to some and at the same time indifference, or even disbelief to others. Curiously, an identical phenomenon affects Logic Programming, namely programming languages such as *GNU Prolog* [DC00]. Many see logic programming as an important and powerful tool that can be used successfully in a wide variety of situations, but for others it is only suitable for academic projects associated with artificial intelligence research.

The work presented in this thesis represents an effort to join together these two *love-hate* concepts/technologies by implementing an information system capable of processing and reasoning over Web ontologies by means of

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

contextual logic programming.

> *I have a dream for the Web [in which computers] become capable of analyzing all the data on the Web [...]. A "Semantic Web", which should make this possible, has yet to emerge, but when it does, the day-to-day mechanisms of trade, bureaucracy and our daily lives will be handled by machines talking to machines. The "intelligent agents" people have touted for ages will finally materialize.* – Tim Berners-Lee, 1999.

As envisioned by Tim Berners-Lee, the Semantic Web will bring to life software agents capable of dealing with many daily needs in an almost fictional way: computer agents that find the information they need and negotiate with other similar agents in our behalf so they can manage our data, our appointments, our calendar in an automated way. Web searches will become intelligent and more accurate, data will be interchanged by all kinds of Web services and our life will become better. However, the more sceptic claim that the Semantic Web will represent only a little more than what we have now, maybe a little smarter at the cost of extremely complex software.

As both the amount of information and its complexity grows frenetically in our technological networked society, there is a need for smarter and more refined computer support for personal and networked information that has to blend the boundaries between personal and group data.

To reach the level of functionality required in such a vision, a few technologies must emerge and mature, for instance:

- A data format of machine-readable documents with meta-data;
- Languages to allow the presentation of that kind of data;
- Languages for querying these;
- Sharing and interchanging of data by universal Web services;
- A trusted and secure layer for data sharing;
- Appropriate software agents;

The fact is that those technologies already exist and there is no need to wait for new revolutionary technologies. For instance, we already have Web

ontologies expressed as RDF/S (Resource Description Framework Schema) and OWL files and we have SPARQL and other query languages. The hard work now is to improve those languages and protocols for standardization purposes in order to allow an evolution from extensions of what already exists to the emergence of advanced software agents that integrates logic, semantics and artificial intelligence.

1.2 Motivation

The Semantic Web concept is recent. Although it was first idealized by Tim-Berners Lee around 1994 when he founded the *W3C - World Wide Web Consortium*, it was in recent years that his vision has began to gain some momentum. W3C is currently the main international standards organization for the World Wide Web and is organized as a consortium where member organizations maintain full-time staff for the purpose of working together in the development of standards. They host a Semantic Web *FAQ* (Frequently Asked Questions) where the following question and associated answer can be found:

> **1.3 What is the *killer application* for the Semantic Web?**
>
> *It is difficult to predict what a killer application is for a specific technology, and the prediction is often erroneous. That said, the integration of currently unbound and independent silos of data in a coherent application is certainly a good candidate.*

The killer application idea is one of the thoughts that serve as a guide when doing research. It is an appealing quest and presents itself as a desirable goal.

As stated in the *FAQ*, integration of any kinds of data from any kind of source in a single silo where it can be merged and used by several applications represent one of the most dearly wanted concepts in the Semantic Web.

This *killer application* scenario represent the main purpose and motivation for the work presented in this thesis, as it is its intention to create the foundations and first steps of a Semantic Web information system with capabilities for successfully responding these demands.

1.3 Objectives

Our main goal is to use contextual logic programming as a framework for Semantic Web agents, in which knowledge representation and reasoning for

documents described by ontologies can be carried out. As such, we adopted the GNU Prolog/CX framework partly described in [AD03] which makes use of persistence and program structuring through the use of contexts [AN06].

Throughout this thesis, a prototype implementation of a Semantic Web system is described, comprised of two main components:

1. A core that is capable of representing and reasoning over Web ontologies from the perspective of contextual logic programming;

2. A back-end capable of mapping Prolog/CX queries to SPARQL queries, thereby able to query external Semantic Web agents, returning the results as bindings for logic variables present in a GNU Prolog/CX program;

The presented system is also meant to be a foundation for a larger framework, setting sights on an information computational hub for transparent reasoning over data coming from several different sources in any kind of format.

1.4 Related Work

Semantic Web, Logic Programming and Web ontologies are the three main topics to which the presented work relates. The implemented system is meant to act as a Semantic Web system information agent, capable of communicating with other Semantic Web agents as a way for data and knowledge interchange.

Since its emergence, the Semantic Web idea has been clearly exposed both in goals and vision. However, many decisions are yet to be made and many problems and issues remain to be resolved. Among others, capabilities for querying and data interchanging in the Semantic Web are two crucial steps towards the success of the vision. These two steps represent the main scope for the presented work. Although a few different approaches to this topic already exist, we propose a contribution that focus on a different point of view of the problem. Some tools for data access information that already exist are, for instance:

- **Thea** [Van06] - An OWL tool capable of parsing an OWL ontology and representing [Van07] it using Logic Programming;

- **Racer** [Sof07] - An OWL reasoner and inference engine for the Semantic Web;
- **Protege** [Pro06] - A Semantic Web platform that provides tools to construct Web ontologies. It has a plug-in interface [KMR04] that allows integration with Web ontology reasoners such as Racer;
- **Jena** [Jen06] - An Open Source Java framework for the Semantic Web. It provides API's for two Semantic Web languages (OWL and RDF) and a SPARQL query engine;
- **Pellet** [SP04] - A reasoner for the OWL DL sub-language;

XPTO and the associated SPARQL back-end are also Semantic Web data access tools. However, they both use contextual logic programming as the mediator framework for Semantic Web agents, in which knowledge representation and reasoning for ontology documents can be conveniently carried out. Modularity in Logic Programming is an old issue in the scientific logic programming community and has been recognized as an important and relevant challenge. GNU Prolog/CX is an implementation of the concepts of contextual logic programming for Prolog. Its view of modularity brings on board the concept of unit - a unitary module, and context - a group of units that represents a goal execution.

Our vision goes beyond a simple computational hub for data directly associated with Semantic Web languages as OWL and RDF and query languages as SPARQL. Integration and reasoning over data from several sources and in different formats is the ultimate goal to achieve, hopefully, in future work.

1.5 Thesis Organization

The remainder of this thesis is organized as follows: chapter 2 introduces the Semantic Web concept and some of the most relevant technologies associated with it. Chapter 3 discusses the technologies used in the implemented system and gives a perspective of the impact that Logic Programming may have in the Semantic Web. The XPTO system is presented and discussed in chapter 4 Next, in chapter 5 the implementation of a XPTO back-end for the SPARQL query language is presented and discussed. Finally, conclusions are drawn and future possibilities are discussed in chapter 6.

Chapter 2

Semantic Web

The purpose of this chapter is to introduce the Semantic Web. As a concept [BL01] firstly envisioned by Internet inventor Tim Berners-Lee, the Semantic Web embraces many ideas, technologies, advantages and issues that are expected to be covered and discussed throughout the following sections. For a more detailed introduction about this topic please consult the referenced bibliography, for instance [Tho04] or [Gri04].

2.1 Today's World Wide Web

There is no question that the Internet and the World Wide Web has been changing our society from many years now. Since its first appearance among the military and academy communities to the present day, the Internet has revolutionized, among many other things, the way we communicate with each other, do business and access information. This technological revolution is leading us toward a knowledge society where the computers, or more importantly, the data and knowledge pool they represent, stand as one of the most important aspects of our lives. Today's Internet is all about information and communication, where us, humans beings, are the source and destination of those technological instances. The majority of the Web pages and Internet applications written are for human readability and comprehension, where computers act as the technological vehicle to achieve that purpose. As of this writing, typical uses of the Web involve people's seeking and making use of information, searching and communicating with other people, purchasing products via electronic commerce and using Web applications. However, these activities are not well supported by software tools, and the only one we can not do without is search, clearly the main tool for using today's Web. Nevertheless, there are some problems [Gri04] associated with its use:

- Lots of pages retrieved, low precision on their relevance with regard to our search. Too much can easily become as bad as too little;
- High sensitivity to vocabulary. Most of the times we do not get the results we want because relevant documents use different terminology from the original query, although semantically queries should return similar results;
- Results are single Web pages, and if we want information that is spread over several documents, we must throw different queries, one at a time;

Despite great technological improvement achieved by most companies with their search engines, the difficulty remains the same: the amount of information and Web content outpaces technological progress, which makes the Web a gigantic unorganized pile of information, where most of it is not reliable.

One of the main obstacles against providing better support to Web users is that, at present, the meaning of Web content is not machine readable. Computers can adeptly parse Web content for layout and routine processing, but in general they have no reliable way to process the semantics. We can see our bank statements, our photographs and our appointments in a calendar, all this on the Web, using some great Web applications. But can we browse our photos in our personal calendar to see what we were doing when we took them? Can we see our bank statement lines in our calendar?

One way to achieve this vision is to struggle for a more intelligent Web, where its contents is more machine processable, thus enabling the use of techniques to take advantage of these representations, which can be described by meta-data that declares what the Web pages are, what they are capable of doing, and how they might change over time.

This vision is called the Semantic Web, mainly propagated by the Word Wide Web Consortium (W3C) [Con07], where the driving force is its director Tim Berners-Lee, the very person who invented the Word Wide Web back in the late eighties. It is important to realize that the Semantic Web will not be a technology parallel to the World Wide Web; instead it will gradually evolve out of the existing Web. From this initiative he expects the realization of its original plan for the Web, a vision where the meaning of information played a more important role than it does in today's Web. In a seminal Scientific American article in May 2001, Berners-Lee and his colleagues explain:

> *"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."*

The above statement clearly tells a few of the most important facts about the Semantic Web: its emergence, not as a new Internet but as an extension of the current Web, the importance of *meaning* in information published on the Web and the desire for capabilities to allow a standardized global access to data, in any format from any data source. Moreover, what ideally describes the Semantic Web vision is to add a meta-data layer on the current Web so that it can be shared, trusted and consumed by software agents in order to proceed with their automated actions.

2.2 Evolution Towards the Semantic Web

As the originator and mentor of this vision Tim Berners-Lee puts it [BL01], the Semantic Web is a natural evolution of the Internet and, hopefully, will provide the foundations for the emergence of intelligent systems and agent layers over the World Wide Web. As said, the standard Web page provides data oriented for human comprehension, which means a computer agent can not intelligently reason about that information, and therefore, can not act as an intelligent tool for a personal computer user. The Semantic Web strives the creation of information technology that will allow explicit machine-processable meta-data documents that describe the meaning and semantics of the data published in the Web.

All this means a revolutionary move in some important areas as the Knowledge Management topic, which is, nowadays, one major organizational issue present on any competitive company and organization. Most information is currently available in a weakly structured form, and with the arrival of the Semantic Web, much more advanced knowledge management systems can be achieved: knowledge can be organized in conceptual silos according to its meaning, query answering over multiple documents will replace keyword-based search and maintenance can be supported by automated tools, which will be capable of checking for inconsistencies and extract new knowledge.

The Semantic Web is the Internet of structured and organized information. This structure will make it easier to perform actions over the Web, like execute complex tasks built upon the results of knowledge reasoning.

2.2.1 Search and the Semantic Web

As mentioned, current keyword search still has some weaknesses and limitations. Can the implementation of the Semantic Web progress towards the

perfect Web search? As John Battelle states is his book *The Search* [Bat05], perfect search will require more than ubiquity and personalization. The vast information now available is often meaningless unless it is somehow tagged and organized. In 2002, Paul Ford wrote an essay entitled *August 2009: How Google beat Amazon and eBay to the Semantic Web* [1] that tied together Tim Berners-Lee vision and the then-emerging power of Google:

> *"Enter Google. By 2002, it was the search engine, and its ad sales were picking up. At the same time, the concept of the "Semantic Web," was gaining a little traction [...]. So what's the Semantic Web? At its heart, it is just a way to describe things in a way that a computer can "understand." Of course, what's going on is not understanding, but logic, like you learn in high school."*

Ford did not stop here and showed how, once the Semantic Web arrived, Google would explode into a global marketplace that would tie together good information about sale products and good search engines connecting them. It is impossible to say that the Semantic Web will bring upon us the perfect information search engine, but it will certainly bring very powerful foundations for better ones. For instance, the search engines will be able to look for pages that refer to a precise concept instead of collecting all the pages in which certain and ambiguous keywords occur. Moreover, the search engine will be able to explore the generalization or specialization of information. When a query fails to retrieve relevant documents about the asked query, the search engine may suggest a more general query, or in the case where there are thousands of hits, a more specific one. The search engine can also be more proactive is this matter and run this *follow up* queries by itself.

2.2.2 Semantic Web vs Web 2.0

Another currently hot Internet topic is the so called Web 2.0. Considering all that was written in this chapter about the Semantic Web and its role as an evolution of the current Web, the reader might understandably think the tag *Web 2.0* fits as a natural name for the Semantic Web, and then assume that both are the same concept. They are not, although at the time of this writing this still represents a common misinterpretation.

Web 2.0 is all about Web applications, great looking Web design and, more importantly, Web community. This means Web 2.0 is a way of building

[1] The complete essay is available at http://www.ftrain.com/google_takes_all.html

services over the World Wide Web universe, using existing technology, but following some straight design principles. Author Paul Graham [2] described in an essay called *Web 2.0* [3] three representative Web 2.0 guidelines:

1. **Technology**. Web 2.0 applications function much like any desktop one, which basically means a Web application can now be more than a simple collection of pages linked together. This is achieved using good DOM [4] manipulation and *AJAX*, shorthand for *Asynchronous JavaScript and XML*.

2. **Democracy.** Web 2.0 is about community and information sharing. Sites like *Wikipedia*, *Digg*, *Del.icio.us* and *Reddit* have revolutionized what most people decided that count as news and where to we look for information and knowledge.

3. **Don't mistreat users.** If a Web page tends to be an application, it must have a very good level of usability and a quality human-machine interface, but more important it must know how to treat its users well. During the Internet bubble a lot of popular sites were just too bad, loaded with obtrusive elements that sent the message that this was *their* site, not the user's site.

All that said, it is important to underline that Web 2.0 is not technology, it is not AJAX, and it is not a features list. It is a concept without hard boundaries, a set of design principles and practices. The Web 2.0 concept began in a conference brainstorming session between O'Reilly and MediaLive International, with the purpose of studying what all the companies that have survived the collapse of the big Internet bubble have in common. Later, Tim O'Reilly wrote an essey entitled *What is Web 2.0* [5] in which he explains his vision about design patterns and business models for the next Internet based generation of software.

Web 2.0 is all about people: let the users create, collaborate, share and interact despite whatever back ends are used and how they work. The Semantic Web is, in a way, on the opposite end: standardize all your data in

[2]Paul Graham is an essayist, programmer, and programming language designer. He founded the companies *ViaWeb* - later acquired by Yahoo and *Y Combinator*. He is also the author of the famous *Hackers & Painters* book.

[3]http://www.paulgraham.com/Web20.html

[4]The Document Object Model (abbreviated DOM) is a tree-like representation of the HTML in the page. Using Javascript, one can manipulate page elements on the fly

[5]http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-Web-20.html

one technology, encode it in another and let the machines loose on it. So, we can think of Web 2.0 as high-level experience directly aimed for the personal computer user and the Semantic Web as a low-level data solution. But the great thing is how these two paths can interact. Tim Berners-Lee presented in a talk [6] his opinion about this matter and concluded with the idea that both trends are good separately and great together. Kendall Clark, managing editor of *XML.com* and the managing principal of *Clark & Parsia LLC*, a Semantic Web company, presented one possible scenario:

> *"Imagine being able to ask Flickr [7] whether there is a picture that matches some arbitrary set of constraints; if so, then asking delicious whether it has any URLs with the same tag; finally, turning the results of those two distributed queries into an RSS 1.0 feed. "*

This and other scenarios are very promising and not too far from happening. Web 2.0 is already a reality present in many Web sites, and much more are being built at this time. With the emergence of the Semantic Web, it is only a question of time before scenarios like the one described above start popping around the Web.

2.3 Semantic Web Technologies

Billy woke up and stood against the bathroom mirror. As he looked at his cheek, he could understand the growing pain he was feeling. He opened his laptop and begun to talk with Beatrix (a Semantic Web agent software): - I want to make a dentist appointment for next week. Beatrix began her duty by checking Billy's last dentist appointment. However, this is the first time in years that Billy needs this type of checkout, so no luck there. Beatrix retrieved recommendations details provided by Billy's primary care doctor agent and fetched a dental care office list. The first hit was Adam & Adam Facilities, and communication with his agent began. However, their schedule didn't have any available slot in the upcoming week. The second hit, Milo Dental Health, was about fifteen miles away from work, was available, but its reputation was low in the trusted rating services. The next hit was CTU Dental Care, and Beatrix tried to match available appointment times with Billy's calendar. In a few minutes Beatrix returned two proposals, but unfortunately Billy was not happy with either of them. Billy decided to set stricter time constraints and

[6] The talk is available at http://www.w3.org/2006/Talks/1108-swui-tbl/

[7] Flickr is a Web application (http://www.flickr.com) for publishing and sharing photos.

asked Beatrix to try again. The next alternative was a dental care office with great reputation, only a few miles away from Billy's work office, and an open time slot that made Billy happy. However, Beatrix was installed only a few days back, and so Billy asked Beatrix to explain some of her decisions. Beatrix provided appropriate information. Billy was satisfied, and asked Beatrix to take all the necessary steps to finalize the task.

The personal agent future scenario presented above is not science fiction; it does not require new revolutionary technology or an outrageous scientific progress to be achieved. The challenge [Gri04] is more one of technology adoption rather than a scientific one. Partial solutions to all of the important problem parts exist, where integration, standardization, adoption by users and development of tools is what is most needed at this point. But, of course, further advances in technology will lead to a more advanced Semantic Web and will enable to step over the present difficulties.

Data About Data

The Web pages we can see today in the Internet are predominantly written in HTML. This will present the information in a way that is acceptable for people to read and understand, but unorganized and meaningless for computer based engines. To step over this issue, the Semantic Web focuses on adding information to Web pages: meta-data, information about information. If HTML is replaced by a more appropriate language, then the standard Web pages could carry alongside their content information that brings meaning to its information, data about data that captures the meaning of the data present on the Web page.

2.3.1 Semantic Web Main Principles

Standardizing on the key technological components that enable the development of the Semantic Web is one of the main goals of the Word Wide Web Consortium for the Semantic Web. Around the end of 2001, Marja-Riitta Koivunen and Eric Miller from W3 published a document [8] describing six main Semantic Web principles. Those are detailed next and viewed as major steps and guidelines for the Semantic Web development:

1. **Everything can be identified by URIs.** An identifier is needed to unequivocally name an instance or resource. Fortunately, the Web already has this concept: the URI (Uniform Resource Identifier). This

[8] The document is available at http://www.w3.org/2001/12/semWeb-fin/w3csw

gives the possibility of referring to an object with an identifier overcoming problems related to different geographic locations. Thus, for example, the city of Évora can be referred to by the URI of its official Web page (Figure 2.1).

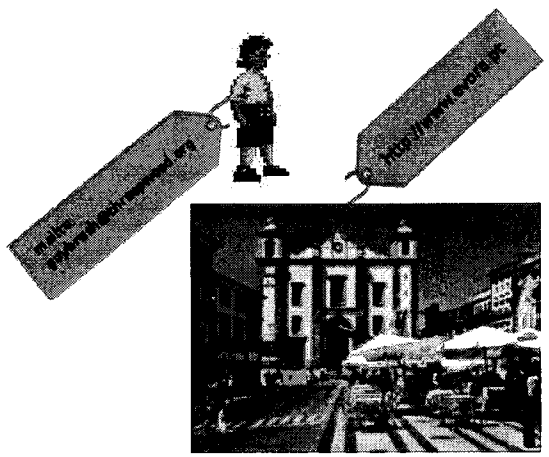

Figure 2.1: Semantic Web identifiers

2. **Resources and links can have types.** The current Web consists of resources and links that represent the addresses of those resources. Although a human can easily distinguish a Dental Web service from Mike's personal Web page, a machine will have some problems. Therefore, each resource can have associated types to explicitly specify if it is a document, a file, a person, etc.

3. **Partial information is tolerated.** Like the current Web, the Semantic Web is unbounded. Semantic Web tools need to tolerate the data decay that comes from problems like linked resources that cease to exist or addresses that may be reused and still be able to function in spite of that.

4. **There is no need for absolute truth.** Not everything found on the Web is true and the Semantic Web does not change that in any way. The applications can decide what they trust and what they do not by using the context of the statement, like who said what and when and what credentials they had for saying it.

5. **Evolution is supported.** Every Web defined concept has one or more authors. However, each author can define the same concept in different ways at different times. The Semantic Web aims to allow the combination of all these definitions and not discard any past information (See Figure 2.2). So, it uses descriptive conventions that can expand as human understanding expands. In addition, the conventions allow effective combination of the independent work of diverse communities even when they use different vocabularies.

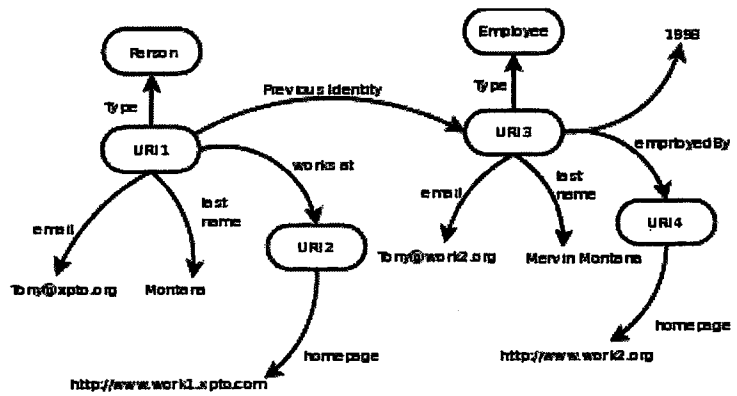

Figure 2.2: Objects evolution

6. **Minimalist design.** A major important design goal is to make the simple things simple and the complex things possible, aiming to standardize no more than is necessary.

2.3.2 Semantic Web: a Layered View

Tim Berners-Lee's view and design of the Semantic Web proceeds in steps, each one building a layer on top of the other. This vision is represented in Figure 2.3, which presents the classic layered Semantic Web scheme. This approach is meant to simplify and ease the consensus over discussions about each component, since usually there are several research groups moving in different directions. In building one layer on top of another, two important design principles should be followed:

- **Downward compatibility.** Agents that fully understand a layer should also be able to interpret information written in lower layers.

- **Upward partial understanding.** Agents that fully understand a layer should also be able to partially interpret higher layers.

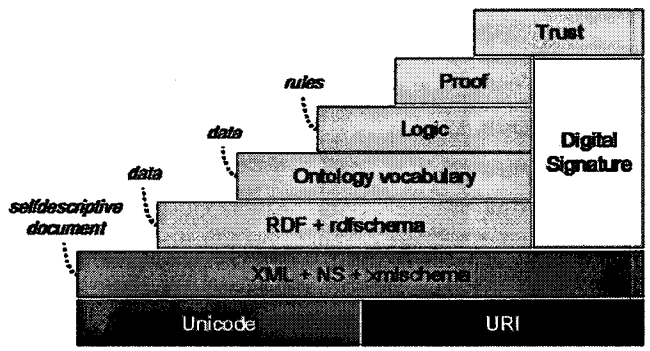

Figure 2.3: Semantic Web layers

At the bottom we find the *URI* and *Unicode* layers. Those are meant to make sure everyone uses international character sets and provide means for identify objects in the Semantic Web. The *XML* layer makes sure we can integrate the Semantic Web definitions with the other XML based standards. XML (Extensible Markup Language) boosts the functionality of Web documents exchange by providing structure and means to identify information in a flexible and adaptable way.

Above is the *RDF* and *RDF-S* [MM04] layer. RDF (Resource Description Framework) is a basic data model for describing simple statements about objects and, although RDF does not rely on XML, it commonly has an XML-based syntax, and therefore is commonly called RDF/XML [9]. RDF Schema [MM04], which is based on RDF, provides additional modeling primitives like classes and properties that enable the hierarchical organization of Web documents. RDF-S is a language for the description of resources and their types and can be viewed as a limited and primitive language for describing ontologies. Next is the *Ontology* layer. It supports the evolution of vocabularies as it can define relations between different concepts. Plus the *Digital Signature* layer for detecting changes to documents, these are the layers that are currently being standardized in W3C working groups.

On top of the Ontology layer sits the *Logic*, *Proof* and *Trust* layers. Those are currently being researched and only simple application demonstrations

[9] As a data model, RDF assumes several forms. Here we present it as XML documents, bu it can also be presented in other formats like a *Turtle* serialization[Dav07].

are being built. The purpose of the *Logic* layer is to enable the writing of rules while the *Proof* layer executes rules and evaluates together with the *Trust* layer for applications whether to trust the given proof or not. Most of the work today is happening around the *Ontology* layer, and as already observed [10] by Tim Berners-Lee, the higher layers are likely to take place only in a near future.

Since its first publication, the layered approach has been used as one of the main references for explaining the Semantic Web architecture. However, as time went on it has suffered several changes resulting of the constant development of the Semantic Web technologies. At the time of this writing, the latest version is illustrated in Figure 2.4.

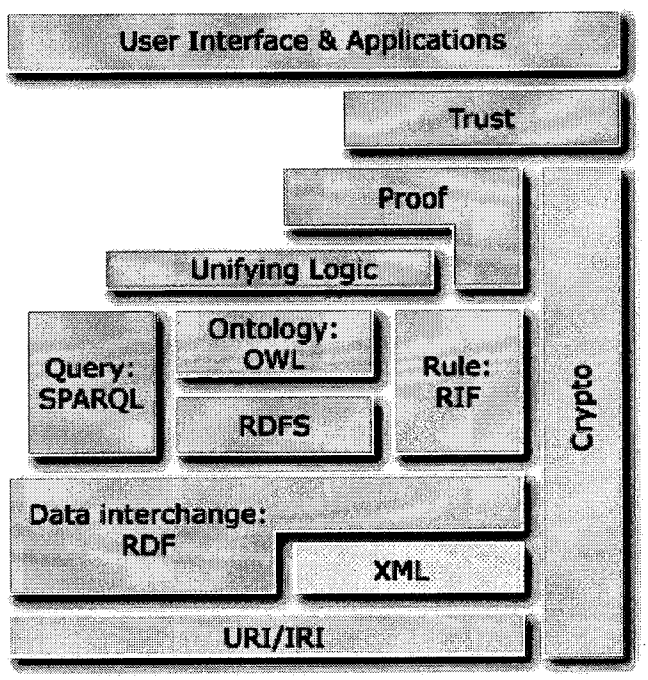

Figure 2.4: Semantic Web layers, latest version

Note that *Rules* and the *SPARQL* query language are now mentioned, which points out the fact that how the Semantic Web data is going to be queried is a very important aspect. Also note the *Applications* layer at the top. Semantic Web applications could be really important since they could

[10]The article is available at http://www.xml.com/pub/a/2000/12/xml2000/timbl.html

help create tension among software developers for convergence of Semantic Web vocabularies. The Semantic Web stack is not written in stone and probably will suffer more changes in the future. For instance, authors Bijan Parsia and Ian Horrocks discuss in more detail the Semantic Web architecture and also introduce an alternative two towers approach [HPPSH05].

As Figures 2.3 and 2.4 show, RDF is the main language for representing and interchanging information. Note that information is not data. Data is what XML represents in a document that, for example, is sent from point A to point B. That data will probably have no use outside those two parties, whereas RDF is designed to present information to be shared, published and used by anybody. This information is transported as triples in the form *(Subject, Property, Object)*.

RDF, as a data model, can be expressed in several ways. Being the underlying structure of any expression in RDF a collection of triples, those can be better represented as a graph for human understanding. However, the Semantic Web requires machine accessible and processable representations, and therefore W3C developed a XML syntax described in [KC07]. Figure 2.5 shows an example.

```
1  <rdf:Description
2       rdf:about="http://www.xpto.org/hattori_hanzo">
3     <ex:starred_in>
4       <ex:movie rdf:about="http://www.xpto.org/kill_bill" />
5     </ex:starred_in>
6  </rdf:Description>
7  <rdf:Description
8        rdf:about="http://www.xpto.org/pulp_fiction">
9    <ex:similar_genre
10       rdf:resource="http://www.xpto.org/reservoir_dogs" />
11 </rdf:Description>
12 </rdf:RDF>
```

Figure 2.5: RDF syntax

Another used RDF syntax is the Turtle [Dav07] representation. Not as complex as the XML serialization and more complete that the graph triples view (it allows the use of prefixes), it is a great RDF representation for pedagogical purposes. Figure 2.6 shows the same graph presented in Figure 2.5,

this time in Turtle notation.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix ex: <http://www.xpto.org/> .

ex:hattori_hanzo ex:starred_in ex:kill_bill .
ex:kill_bill rdf:type ex:movie .
ex:pulp_fiction ex:similar_genre ex:reservoir_dogs .
```

Figure 2.6: Turtle syntax

Note that RDF is not the only data model that can be useful in the Semantic Web vision: for instance, *Topic Maps* is another important data model for representation and interchange of knowledge. Topic Maps was created to support high-level indexing of sets of information resources in order to make information easily findable. RDF was intended to support the vision of the Semantic Web by providing structured meta-data about resources. The two have significant conceptual differences although they share a central objective: define a format for the exchange of knowledge on the Web. Since both seem to have well established communities, it is fundamental that they can integrate with each other in order to prevent a partition of the Web into collections of incompatible resources. Consult [LD01] for more details about this issue.

2.3.3 Web Ontologies

Merriam-Webster [11] dictionary defines ontology as: *"1- a branch of metaphysics concerned with the nature and relations of being; 2- a particular theory about the nature of being or the kinds of things that have existence"* . Although this definition introduces ontology as a term originated from philosophy, the word *ontology* has become one of many that have been given a Computer Science technical meaning different from the original one.

In the Computer Science context, an ontology is a data model that represents a set of concepts within a particular domain and the relationships between those concepts. For Web pages, this means it provides a shared understanding of the Web page domain. This is an important concept since problems could arise because of different terminology usage. A zip code in

[11] http://www.m-w.com/dictionary/

one application may be the area code in another. Thus, mapping a particular terminology to a shared ontology will enable semantic interoperability between ontologies.

Generally, an ontology consists of a finite set of terms and the relationship between those terms. These terms denote some important concepts such as:

- Classes of objects (instances);
- Relationships between classes (hierarchy);
- Relationships between objects;
- Properties;
- Value restrictions;
- Disjointness statements;

Figure 2.7 illustrates a simple ontology (introduced in [TF06]), represented as a graph, that describes a simple hierarchy of the books category *Writing, Novel, Essay, Historical Novel, Historical Essay*, and the two books *The First Man in Rome* and *Bellum Civile.*

For example, we can see that *Historical_Novel* is both a *Novel* and an *Essay* and that books may optionally have translators, as is the case with *Bellium Civile.* Books, authors and translators are represented by nodes without identifiers called `blank nodes` and the only assumptions are based on the `subClassOf` and `instance` relations.

2.4 Web Ontology Languages

In order for ontology documents to start being used, two kinds of technologies must rise: description and query languages for ontologies. Together these will allow not only the description of domains in a uniform manner but also a way to retrieve information from those domains.

2.4.1 Ontology Description Languages

There are languages for representing information in the Semantic Web, for instance: RDF, RDF-S and OWL. RDF is the underlying language that is capable of represent very simple information. RDF-S extends RDF, resulting in a very simple ontology language. It has the same constructs that RDF

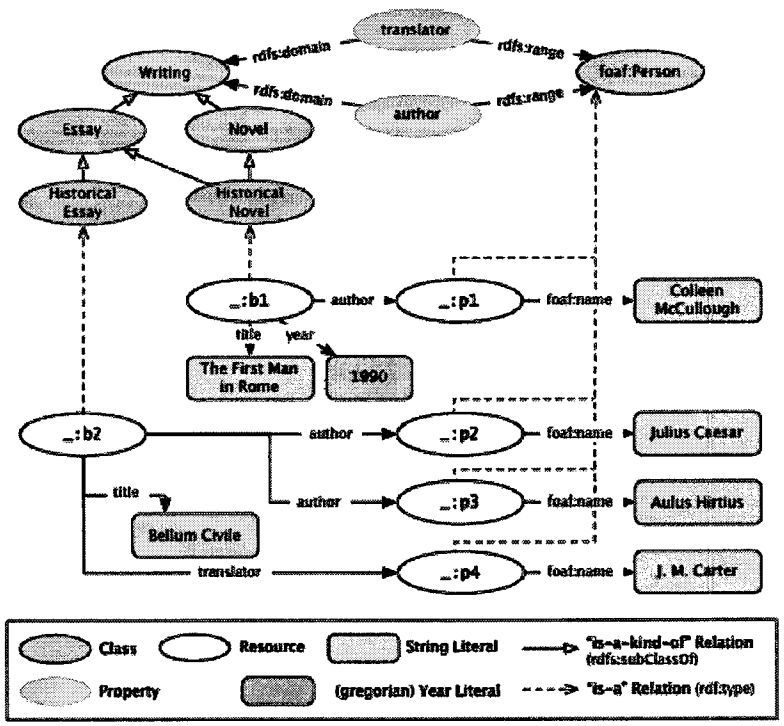

Figure 2.7: Books ontology example

has, but it adds some important built in properties for defining relationships.

However, the *Web Ontology Working Group* of W3C identified a number of characteristics and use-cases for the Semantic Web that would require more expressiveness than RDF and RDF-S can offer:

- **Local constraints.** For example, RDF-S does not have direct means for stating that *parents of people are parents.*
- **Conjunction.** It is not possible to make thinks like *movies with at least two producers.*
- **Definitions.** Definitions like *A Movie actor is a male or female person with at least one movie participation* also can not be represented directly in RDF-S.

A richer ontology modeling language was already defined, DAML-OIL [DAR07], which was then taken as the starting point for the W3C Web

Ontology Working Group in defining OWL [MvH05]. It has all the characteristics needed to fit into the Semantic Web: it uses URIs for names, information is also represented as RDF triples and it is semantically compatible with RDF and RDF-S. However, OWL lays down a tradeoff between expressive power and computational difficulty because inference in OWL can actually be undecidable. The need for restricting OWL became clear and so OWL was divided into three species: OWL Lite, OWL DL and OWL Full.

OWL Full contains all the OWL language constructs, it is meant for users who want maximum expressiveness, but offers no computational guarantees, i.e., not all conclusions are guaranteed to be computable and finished in finite time. OWL DL is a sub-language of OWL which introduces some constraints on the OWL language usage: these provide a maximal subset of OWL against which there are guarantees that any query is decidable. OWL Lite provides the basics for subclass hierarchy construction: subclasses and property restrictions. The idea behind the Lite expressiveness limitations is that they provide a minimally useful subset of language features that are straightforward for tool developers to support.

OWL: Classes, Properties and Instances

An OWL document has classes and instances of those classes, where a class describes a set of objects with a set of properties and relations with other objects. OWL documents are RDF documents and may start with a collection of assertions that forms its header. Those assertions may contain information like comments, version control and inclusion of other ontologies.

Classes are defined using an `owl:Class` element and, by default, all of them are instances of the predefined class `owl:Thing`. For example, we can define a class `Actor` as illustrated in Figure 2.8.

```
1 <owl:Class rdf:ID="Actor">
2         <rdfs:subClassOf rdf:resource="#Person" />
3 </owl:Class>
```

Figure 2.8: OWL class

In OWL there are two kinds of properties:

1. ***datatype* properties.** Relations between a class instance and a literal

or datatype value. Note that OWL does not have any predefined data types. Instead, it allows the usage of XML Schema datatypes.

2. ***object* properties.** Relations between two classes.

Figure 2.9 shows an example of an object property and a datatype property:

```
<owl:ObjectProperty rdf:ID="isDirectedBy">
     <rdfs:domain rdf:resource="#movie" />
     <rdfs:range rdf:resource="#filmDirector" />
</owl:ObjectProperty>

<owl:DatatypeProperty rdf:ID="Movie genre">
     <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>
```

Figure 2.9: OWL object and datatype properties

Instances of classes (individuals) in OWL can be declared as in RDF or using a clear syntax provided by OWL. Figure 2.10 shows an example of both declarations of the *Taxi Driver* movie.

```

<rdf:Description rdf:ID="taxi driver">
     <rdf:type rdf:resource="#Movie"/>
     <director rdf:resource="#Martin Scorcese" />
     <genre rdf:datatype="&xsd;string">drama,crime</genre>
</rdf:Description>

<Movie rdf:ID="taxi driver">
     <director rdf:resource="#Martin Scorcese" />
     <genre rdf:datatype="&xsd;string">drama,crime </genre>
</Movie>
```

Figure 2.10: OWL instances example

The examples above represent only a brief overview of the OWL language. For further details about the language, please consult the W3C OWL speci-

fications that are spread among six documents.

At the time of this writing, OWL 1.1 is on its way (see [PSH07]). This is an OWL extension that is meant to bring a small but useful set of features that have been requested by users, for which effective reasoning algorithms are now available, and that OWL tool developers are willing to support.

2.4.2 Ontology Querying

An open Semantic Web research issue has been the lack of a standard query language. RDF query languages have been in discussion at W3C since the QL'98 [12] workshop in December 1998. Development of XML querying started around 1999 (XQuery) and since then its developers have also been leading research about RDF querying languages. W3C wrote the *RDF Data Access Use Cases and Requirements* [Cla07] where it was recorded some use cases for RDF:

- Finding values for partially known graph structures;
- Getting information about an identifiable object with unknown properties;
- A human friendly syntax for queries for application developers;
- Running automated regular queries against RDF graphs;
- Querying aggregated RDF graphs;
- Running queries constrained with datatype expressions;
- Querying a remote RDF server and getting streaming results back;
- Allowing alternate solutions to match in queries;

Query answering on the Semantic Web is complex, even more complex than on the traditional Web because *meaning* must be properly understood and processed. There are some different proposals and approaches for this kind of query languages, as stated by the research presented by Tim Fuche and Bry [TF06], ranging from pure *selection languages* with limited expressivity to general purpose languages supporting different data representation formats and complex queries. However, as of this writing, only RDF query languages are already in use, whereas other data modeling formalisms such as

[12] http://www.w3.org/TandS/QL/QL98/

OWL are still an open research issue, and only a very small and incomplete number of proposals [13] for querying Semantic Web data modeled after formalisms other than RDF exists. Fuche and Bry presented a survey [BBFS05] about RDF query languages, where they divided them into three groups:

1. Relational or pattern-based;
2. Reactive rule;
3. Navigational access;

Languages following the first topic paradigm use selection constructs similar to selection-projection-join, much like *SQL* does for relational databases; here we can find languages like *RQL* and *Xcerpt*. The second group uses, as the name implies, reactive rules, but otherwise act very much like the first group. For example, *Algae* is a language that uses reactive rules. Languages that belong to the final group use navigational access and path expressions over patterns.

SPARQL

At the present day, one specific RDF query language is starting to gain some momentum: SPARQL [PS06]. Its name is a recursive acronym that stands for *SPARQL Protocol and RDF Query Language*. It is undergoing standardization by the RDF Data Access Working Group (DAWG) [14] of the World Wide Web Consortium. Towards the status of W3C recommendation, it was released as a Candidate Recommendation in April 2006, but returned to Working Draft status in October 2006, due to two open issues [15]. Even Tim Berners-Lee stated [16] last year that the emergence of SPARQL will make a big difference around the Semantic Web, since the ability to correctly querying ontology documents is one major Semantic Web goal.

The SPARQL query language consists of the syntax and semantics for asking and answering queries against RDF graphs in a way much similar to SQL. SPARQL contains capabilities for querying by triple patterns, conjunctions, disjunctions, and optional patterns. Results of SPARQL queries can be ordered, limited and offset in number, and presented in several different forms.

[13] One known proposal is OWL-QL, a project lead by Stanford University - http://www-ksl.stanford.edu/projects/owl-ql/

[14] http://www.w3.org/2001/sw/DataAccess/

[15] http://www.w3.org/TR/2006/WD-rdf-sparql-query-20061004/

[16] This statement can be found in a 2006 article called *Berners-Lee looks for Web's big leap* in http://news.zdnet.co.uk/internet/0,1000000097,39270671,00.htm?r=1

Chapter 3

Logic Programming on the Semantic Web

This chapter aims to introduce the reader to Logic from a Computer Science point of view, to discuss why Logic Programming can be an important ally for the Semantic Web and to introduce the GNU Prolog/CX framework used in our work.

In Computer Science, Logic and Logic Programming represent wide-ranging topics that have been the target of research and several use cases from many years now. The objective of this chapter is only to give a brief introduction to these topics, for a more detailed introduction please consult the referenced bibliography, for instance [Joh97] [Baa02] [Baa99] or [Ant90].

3.1 Logic in Computer Science

Enrico Franconi [Fra02] states that Logic in Computer Science can be described as the discipline that studies the principles of reasoning, used in a way to achieve correct conclusions. An *agent* is an entity that perceives and acts according to an internal declarative body of knowledge. A *logic* allows the representation in axioms of a particular domain and the drawing of conclusions from the information contained on that domain. Important characteristics are:

- **Expressiveness**: capable of representing a problem;
- **Correctness**: no false conclusions are drawn;
- **Completeness**: all correct conclusions are drawn;

- **Decidability**: there exists a terminating algorithm to compute entailment (logical implication);
- **Complexity**: what resources are needed for computing the solution;

There exists several types of logic [Fra02] and each one can be characterized by what they commit as *primitives*. Generally speaking, logic provides [Gri04] three main characteristics:

1. Formal notation (language);
2. Formal semantics;
3. How implicit knowledge is made explicit;

Each logic has a syntax, a semantics and an inference procedure. Syntax describes how to write correct sentences, the semantics tells what a sentence means and the inference procedure derives results logically implied by the premises. It indicates that automated reasoners can infer and deduce conclusions from the given knowledge. This knowledge, or a Knowledge Base (KB), is a logic theory, i.e, a set of sentences in a formal language.

Antoniou and Van Harmelen [Gri04] present logic as the foundation of knowledge representation and point a few reasons for the importance of logic in Computer Science:

- It provides a high level language in which knowledge can be expressed;
- It has a well-understood formal semantics;
- There exist proof systems that can derive statements from a set of previously stated premises;
- Proof systems should be *sound* - all derived statements follow semantically from the premises;
- Proof systems should be *complete* - all logic consequences of the premises can be derived from the proof system;
- It should be possible to trace back the proof that leads to a logic consequence;

For instance, Logic Programming is based on *First-order Predicate Logic*, where a both sound and complete proof system does exist. By a result [1] of logician Kurt Gödel, more expressive classic logics - called *high-order logics* do not admit sound and complete proof systems.

3.1.1 First-order Predicate Logic

The base of Predicate Logic is Propositional Logic. Logics are characterized by what they commit to as *primitives* and what is to be believed about facts. Propositional Logic [Fra02] defines facts with three levels of beliefs by an agent: *true, false* and *unknown.* An *entailment* is a logic implication of a knowledge base: *knowledge base KB entails the sentence s if and only if s is true in all worlds where KB is true.* The basic building block of Propositional Logic are atomic statements and logical connectives *and, or* and *not.*

The fact that atomic formulas in Propositional Logic are just statements which may be true of false means there are no internal structure in statements and thus there can be no interpretation of relationship between objects. Let us consider the following statements in Figure 3.1:

```
1 beatrix is female.
2 bill is male.
3 beatrix and bill are married.
```

Figure 3.1: Logic statements

In Propositional Logic (Figure 3.2), the above statements are atomic propositions, whereas in Predicate Logic (Figure 3.3) atomic statements uses predicates with constants as arguments.

```
1 beatrix-is-female.
2 bill-is-male.
3 beatrix-and-bill-are-married.
```

Figure 3.2: Statements in Propositional Logic

[1] For more information consult the *The Work of Kurt Gödel* by *Stephen Cole Kleene*, J. Symb. Log. Journal, 1976.

```
1    female(beatrix).
2    male(bill).
3    married(beatrix, bill).
```

Figure 3.3: Statements in Predicate Logic

Predicate Logic is the term used for symbolic formal systems like First-order Logic or Second-order Logic. Another difference towards other symbolic formal systems is the usage of variables that can be quantified (e.g. existential and universal quantifiers). Predicate Logic (or strictly, First-order Predicate Logic) is an important knowledge representation language. It allows the representation of fairly complex facts about a stated world and the derivation of further facts with guarantees of soundness and completeness: assuming that the initial facts were true then so are the conclusions. A First-order Predicate Logic theory consists of a set of axioms and the statements deducible from them, which is the base for Logic Programming.

Logical reasoning is the process of drawing conclusions from premises using rules of inference. Predicate Logic allows the possibility of reasoning about properties and relationships of individual objects using several inference rules [Fra02]:

- Equivalences;
- Implications;
- Propositional logic inference rules;
- Universal instantiation;
- Universal generalization;
- Existential instantiation;
- Existential generalization;
- Negation;

Sentences in Predicate Logic are built up from atomic sentences, which consist of a predicate name followed by a number of arguments. Each argument is a term, where a term can be a constant symbol, a variable symbol or

a function expression. So, Predicate Logic sentences are constructed by combining atomic sentences with logic connectives and quantifiers. The semantics of Predicate Logic are defined in terms of the truth values of sentences. We can determine the truth value of a sentence if we know the truth values of its basic components. An *interpretation* function determines the truth values of the basic components, given some domain objects that we are concerned with.

One useful proof procedure for Predicate Logic is *resolution.* Resolution is a proof procedure for proving goals by refutation: if a contradiction can be derived from **not P** then **P** must be true. Resolution is a *sound* proof procedure, which means that for something proved with resolution, we can be sure it is a valid conclusion. However, there exist other problems when looking at a proof procedure:

- **Completeness:** it may not be able to always prove something is true even if it is true;
- **Decidability:** the procedure may never finish when trying to prove something that is false (or true);
- **Computational efficiency**;

Note that the efficiency of a proof will often depend as much on how you formulate your problem as on the general proof procedure used.

3.1.2 Description Logics

Description Logics [NB03] emerge as subsets of First-order Logic that describe a family of knowledge representation formalisms. It represents the knowledge of a domain by defining the concepts of that domain and then using these concepts to specify individuals and properties of objects occurring in that domain.

A Description Logic knowledge base can be given semantics by translating it into First-order Logic with equality: atomic concepts are translated into unary predicates, complex concepts into formulas with one free variable and roles into binary predicates. It is based on concepts (or classes) that represent sets of objects, roles (or properties) that represent relationships between objects and individuals representing specific objects. For instance, a concept such as *Person* is atomic. Making use of a set of concept constructors, one can construct complex concepts that describe the conditions of a

concept like *membership*. For example, the concept *hasFather* in Figure 3.4 describes objects that are related through the *hasFather* role with an object from the *Person* concept.

```
1 hasfather.Person
```

Figure 3.4: *hasFather* concept

A Description Logic knowledge base typically consists of a `TBox T` and an `ABox A` [BN03]. The first contains axioms about the general structure of all allowed worlds in the knowledge base and is therefore essentially similar to a database schema. On the other hand, an `ABox` contains axioms that describe the structure of a particular knowledge base world. For example, an axiom that states that each instance of the concept *Person* must be related by the role *hasFather* with an instance of the concept *Person* is a `Tbox` axiom. An `ABox` axiom can state, for example, that *Mike Portnoy* is a *Drummer*.

Nowadays, Description Logic is already an important part of the Semantic Web because of its use in the design of ontologies. Moreover, the OWL sub-languages OWL DL and OWL Lite are based on Description Logic.

3.2 Logic Programming

The most common way to describe logic is through mathematics, thus Logic Programming can be seen as the use of mathematical logic for Computer Science. The use of logic as the basis of programming languages such as Prolog [Iva01] [Ste94] is quite recent, although logic as been used as a tool for many years now in Computer Science. As opposed to the more mainstream programming paradigms, Logic Programming suggests that explicit instructions for operations should not be given, but instead the knowledge about the problem and the assumptions that are sufficient to solve it be stated explicitly, as logic axioms.

Logic Programming is a set of knowledge representation formalisms centered around the notion of *rules* and is appropriate for problem-solving tasks in which two layers can be defined: a declarative representation language and a theorem prover or model generator that is used as the problem solver. The theorem prover is applied to the declarative sentences that have the form of

implications, like illustrated in Figure 3.5 and treats those implications as goal-reduction procedures.

```
1    D1, D2 and...and Dn implies H
2
3    To show/solve/prove H, show/solve/prove D1, D2 and...and Dn.
```

Figure 3.5: Logic programming implications

Numerous variants of these basic formalisms have been considered such as rules with disjunctions in the rule heads or extensions with classical negation. [2]

As stated in [Ste94], a program can be executed by means of a problem definition formalized as a logic statement to be proved called a *goal*. A goal in Logic Programming is proved using Predicate Logic resolution *proof by contradiction* [Gri04], i.e, by negating the goal and proving that a contradiction is obtained using the logic program.

The execution itself is the act to try solving the problem, i.e, to prove the goal given the assumptions in the logic program. The goal statement proof is done constructively. If successful, the unbound individuals provided in the goal are bounded to values, which constitute the output of the computation. When the goal has no variables, the search space for solving the goal is an *and-or tree* [3] determined by the reasoning system where the root of the tree is the goal. Given any node in the tree and any clause whose head matches the node, there exists a set of child nodes which correspond to the sub-goals. These nodes are grouped together by an *and*. The alternative sets that represents alternative ways of solving the node are grouped together by an *or*. The fact that there are alternative ways of executing a logic program has been characterised by the equations [Ste94] in Figure 3.6, whereas the set of axioms represents a program and a computation represents different theorem proving strategies.

Logic Programming focuses on efficient query answering over a bounded data set. Nowadays it is seen as a general problem-solving formalism, capable of succinctly expressing hard computational problems.

[2] For instance, a combination of these two features is commonly known as *answer set programming*.

[3] An and-or tree can be viewed as a parallel execution. It has or-nodes (called choice points) that are created when there are multiple ways to solve a goal and and-nodes that are created when a goal invokes several conjunctive sub-goals.

```
1 program= set of axioms
2 computation= constructive proof of a goal statement from a program
```

Figure 3.6: Logic Programming equations

3.2.1 Contextual Logic Programming

Modularity in Logic Programming is an old polemic concern among the logic programming community and has been recognized as an important and relevant challenge.

The GNU Prolog/CX [AD03] is an implementation of the concepts of Contextual Logic Programming, based on Prolog. One of the main advantages in Contextual Logic Programming is the achievement of modularity, which brings on board the concept of *unit* - a unitary module, and *context* - a group of units that represents a program in which to specify a goal execution.

Contextual Logic Programming (CxLP) is a simple yet powerful extension to the Prolog logic programming language which provides a mechanism for modularity. In CxLP a finite set of Horn clauses with a given name is designated by *unit*. Abreu and Diaz [AD03] provide a revised specification for CxLP, which emphasizes the OOP aspects by means of a stateful model, allowed by the introduction of unit arguments. Informally, a unit is a parametric module, constituting the program's static definition block.

Unit descriptor terms can be instantiated and collected into a list to form a *context*, which can be thought of as a dynamic property of computations. A context specifies the actual *program* (or theory) against which the *current goal* is to be resolved. In short, it specifies the set of predicates which are applicable. These predicates have definitions which depend on the specific units which make up the context. A more extensive description of CxLP may be found in [AD03, AN06].

Some parallels can be made between Contextual Logic Programming (CxLP) and Object Oriented Programming (OOP):

Context and object instance: A (possibly partly) bound context is a list of units which can be described as an object *instance*. There is no true analog for the *class* concept, units being conceptually similar to *components*, although the *context term skeleton* may come close;

Predicate and method: A predicate present in a unit is equivalent to a

method definition in an OO setting;

Goal and message: a goal executed in a particular context can be interpreted as sending a message (the goal) to an object (the context);

Unit argument and instance variable: unit arguments are variables whose scope is the entire unit, much like instance variables in OO;

GNU Prolog/CX introduces a set of language operators called the *context operators* which modulate the context part of a computation.

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 that is the suffix of the C which starts with unit u. Some of the most used operations and operators in GNU Prolog/CX are:[4]

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 query: :< C, unifies C with the current context;

Calling context query: :> C, unifies C with the calling context;

Lazy call: :# G, evaluates the goal G in the calling context;

3.3 Logic for the Semantic Web

Logic, in its reasoning and conclusion drawing form, is likely to play an important role in the Semantic Web. Thomas Passin states [Tho04] that Logic in the Semantic Web is expected to work in the following ways:

- Applying and evaluating rules;
- Inferring facts that aren't explicitly stated;
- Providing capabilities for Semantic Web agents to explain why a conclusion has been reached;

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

- Detecting contradictory statements;
- Representing knowledge;
- Merge information from different sources in a coherent way;
- Create software to execute queries to obtain information of Semantic Web data repositories;

One way that the Semantic Web relates to the above topics is through programs built with a series of `IF-THEN-ELSE` rules, often chained together. A rule processor agent can work backward from one condition to describe what steps were taken to get there. This means it is possible to ask if a certain outcome is possible and how to get to it and therefore simulate a reasoner that has knowledge of a particular domain. For instance, Topic Maps has been used to define sets of rules.

Logic can be used to represent knowledge whereas Web ontology documents supplies the concepts and terms. Logic provides ways to make statements that define, use and reason about those concepts. These collections of statements may come from different data sources such as relational databases, Web pages or other knowledge bases and they may be expressed in many ways such as RDF, OWL or Topic Maps.

In the Semantic Web, Logic will play a different role than many of the other discussed components. Information is to be changed and processed by a program that may in turn obtain more data from other sources spread around the Internet. Ontologies will be shared and merged. However, Logic is presented as a tool to be used and applied to information and not as information to be changed.

3.3.1 Inferring facts from Semantic Web data

Let us see a common example of a simple inference that involves knowledge typically found in ontologies: imagine a faculty domain where all professors are faculty members, all faculty members are staff members and *lucarelli* is a professor. This information is expressed in Figure 3.8, in Predicate Logic.

We can then infer the results shown in Figure 3.8.

The above example illustrates the fact that logic provides a natural way to uncover ontological knowledge that is implicitly given. However, the example is based on simple *if conditions*. More complicated and powerful expressiveness can lead to undecidability barriers, and the more expressive a

```
1  professor(X) -> faculty_member(X)
2  faculty_member(X) -> staff(X)
3  professor(lucarelli)
```

Figure 3.7: Predicate Logic inference example (part 1)

```
1  faculty(lucarelli)
2  staff(lucarelli)
3  professor(X) -> staff(X)
```

Figure 3.8: Predicate Logic inference example (part 2)

logic is, the more computationally expensive deducing conclusions becomes. Nevertheless, the knowledge relevant for the Semantic Web seems to be of a relatively restricted [Gri04] form and is expected to be supported by a wide range of reasoning tools.

One major advantage of logic usage in the Semantic Web is its capability of providing *explanations* for conclusions, i.e, trace all the inference steps. This kind of explanations are very important as they can act as proofs presented in a human readable form, acting as a way for users to increase their confidence in Semantic Web agents. Moreover, explanations will get an important role in agent interactions where one side will have to validate the proof provided by the other. Thomas Passin presents [Tho04] an interesting exemple: a Web shop sends a message to a private agent, via its own agent saying he owes 100 euros. When the latter agent asks for proof, he might get this sequence response from the shop agent:

- Web log that indicates a purchase of product P for 90 euros;
- Delivery cost of 10 euros;
- Proof of delivery: [...];
- Rule from the shop terms:
  purchase(X, P) & price(P, PRICE) & costDelivery(P,Y)
  & delivered(P, X) — owes(X, PRICE+Y);

This example illustrates two important facts: logic must be usable in

conjunction with different sources of data and be machine-processable as well.

3.3.2 OWL Ontologies and Description Logics

A knowledge representation can be viewed as a relating theory to a particular *world* via formal models. An ontology is a particular knowledge base that describes *facts* assumed to be *true* by a *community* of users. This means the power of an ontology, in Computer Science, comes from the fact that it can make domain assumptions explicit and therefore, provide data integration.

Ontology languages allow users to write explicit, formal conceptualizations of domain models. They should have [Gri04]:

- A well-defined syntax;
- Efficient reasoning support;
- An agreed-upon formal semantics;
- Sufficient expressive power;
- Convenience of expression;

RDF and OWL languages can be viewed as specializations of Predicate Logic that provide a syntax that fits its purpose well, i.e, Web languages based on tags. They define reasonable subsets of logic that correspond roughly to a Description Logic, a subset of predicate Logic for which an efficient proof system exits. The OWL language is based on Description Logics [BvHH+05] or, more precisely, on a family of knowledge representation formalisms based on First-order Logic and exhibiting a well understood computational properties. OWL is a family of three ontology languages where the first two are based on different subsets of Description Logics: OWL Lite, OWL DL and the other one, OWL Full, which is meant for cases where maximum expressiveness is wanted with no decidability computational guarantees [SWM04].

OWL compatibility With Logic Programming

The incompatibility of the Open world assumption in OWL with the closed world semantics of Logic Programming has generated some discussion and debate in the Semantic Web community and is one of the main

issues when implementing Semantic Web software based on Logic Programming [MHRS06]. However, this does not mean that Logic Programming cannot be used to reason over OWL Web ontologies, far from it. What it does mean is that there are some practical use cases which are difficult or impossible to realize with OWL but addressable by Logic Programming. Horrocks presents [MHRS06] an overview of how OWL could be integrated with rules without sacrificing semantic compatibility.

3.3.3 Impact of Logic on this Work

The developed system was implemented using Contextual Logic Programming, namely GNU Prolog/CX. The technology used was not based on a typical research and choice among different solutions, but instead the use of Contextual Logic Programming was our motivation whereas the Semantic Web our choice as a target research field to work with.

As already discussed, Logic Programming based agents and software are expected to have an important role on the Semantic Web. They cannot be directly *glued* together and the existing issues on using Logic Programming languages within OWL documents can not be ignored. In the work presented in this thesis, we tried to move forward in order to bring up the advantages of using Contextual Logic Programming to represent and reason over OWL ontologies.

Chapter 4

XPTO - XPTO Prolog Translation for Ontologies

The goal of the XPTO [1] system is to represent Web ontologies from the perspective of Contextual Logic Programming and to enable querying that representation. Web ontologies can be represented with OWL which is sub divided into three sub-languages: OWL Lite, OWL DL and OWL Full.

OWL DL emerged as the goal for the mapping and representation capability by XPTO since the specification of OWL Full does not to guarantee computational completeness nor decidability still guaranteed by the OWL DL language, i.e., that all conclusions are guaranteed to be computable and that all computations will finish in finite time.

In XPTO, ontologies are treated on a *per file* basis. The information represented in the ontology file 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 file is parsed as a plain XML structure, resulting in a Prolog term representing the complete ontology file. This process is described in section 4.1. In the unit generation phase, the Prolog term is transformed into a dictionary, a 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. Next, section 4.3 introduces the manner in which to retrieve the information from the representation. After the information in an ontology is transformed into a set of GNU Prolog/CX units and interface predicates, the possibilities and capabilities of that representation are

[1] This work was developed in cooperation with Nuno Lopes.

equivalent to that of a Prolog program, with the benefit of a modular program structure, i.e, the ontology may be queried as if it were a regular logic program. In section 4.4 some real-life examples of XPTO are introduced and in section 4.5 we compare XPTO with similar systems and present the parser and query benchmark results. Finally, in section 4.6 we draw some conclusions and discuss future work.

4.1 Ontology Parsing

The first step towards building the ontology representation is parsing. 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. 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 4.5.1.

The selected parser was the Expat XML parser [Coo06]. Two main reasons influenced the choice of this parser: the results of the benchmark tests and the easy integration of C and Prolog.

The Expat 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 passed on to Prolog. This term is an accurate representation of the XML file, apart from the possible comments in the XML/RDF/OWL file, there is no further loss of information in this transformation.

4.1.1 Prolog Representation for XML Documents

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

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

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: `ElementAttributesList` is a list of the XML node's attributes in the form `AttributeName = AttributeValue`. AttributeName will also be of the form as `ElementName`.

SubElementsList: `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 (other elements of the same structure) or element values which will be represented by its value.

For example (all the examples presented in this chapter uses the *Wine* OWL DL ontology [W3C06] which is a sample ontology used in the OWL specification documents), this representation will produce the structure represented on Figure 4.2 for the XML code in Figure 4.1.

```
<!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: Vintage Class definition - wine.rdf

4.2 Ontology Mapping

XPTO is prepared to translate ontologies defined in OWL Lite or OWL DL into Prolog. This mapping must allow for easy access to the information

```
[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 class Vintage

represented in the ontology, using standard Prolog goals.

After the ontology is represented by a Prolog term, a dictionary data structure will be generated with the necessary information to later create the GNU Prolog/CX units. These generated units are then compiled and loaded into the running program. Next we describe the process of translating the Prolog term into the incomplete structure and explain the representation of the ontology using GNU Prolog/CX.

4.2.1 Ontology representation

A GNU Prolog/CX unit is a named and parametrized set of Prolog predicates. In XPTO, ontologies are represented using *units* and these will be used to represent not only the whole ontology but also each OWL class and property. This scheme is represented in Figure 4.3.

The information about the ontology is represented in a specific unit named `ontologies`. This unit 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. Further information about each one can be found in its unit. This naming schema for properties and classes does not present a problem in OWL

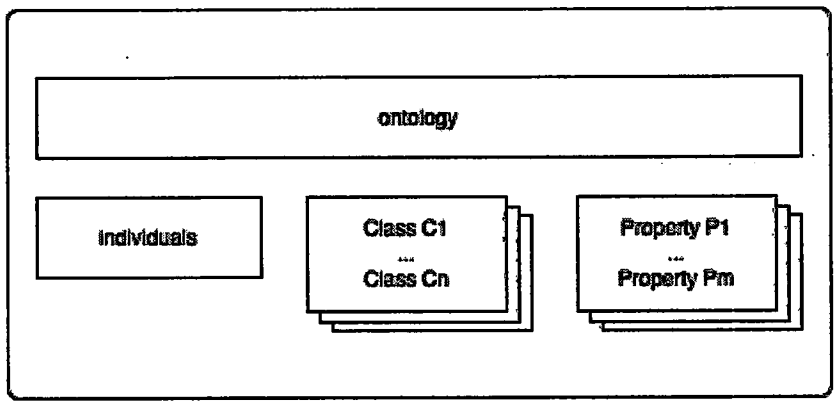

Figure 4.3: Ontology representation schema: units

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, i.e., there could never exist, for instance, a class with the same name as a property."*

The ontology individuals are represented in the unit `individuals`. It contains the name of the individuals, individual relations and class memberships. In the following sections we describe the structure of these units and discuss some alternative representations which we previously experimented with.

`Ontology` 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 the value, given the individual name that shall be retrieved previously from the GNU Prolog/CX context. For example, the definition of a property and its representation are show, respectively, in Figure 4.4 and Figure 4.5. An example of its usage is shown if Figure 4.9 on page 50.

```
<owl:ObjectProperty rdf:ID="locatedIn">
  <rdf:type rdf:resource="&owl;TransitiveProperty" />
  <rdfs:domain rdf:resource="http://www.w3.org/owl#Thing" />
  <rdfs:range rdf:resource="#Region" />
</owl:ObjectProperty>
```

Figure 4.4: LocatedIn property definition - wine.rdf

```
:- unit(locatedIn).

object(rdf : type('TransitiveProperty')).
domain('Thing').
range('Region').
type(object).
```

Figure 4.5: LocatedIn property unit

Class Units

These units will represent each class of the ontology and all information relevant to it: this includes restrictions on the individual properties 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. This process is 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, 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 4.2.3.

`Individuals` Unit

This unit contains all the individuals, their properties and information about individual relations. The individual properties are stored as triples, much 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 individuals 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 for defining 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 [MvH05]. 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 4.2.3.

4.2.2 Alternative representations

One approach to map an ontology that we experimented was to represent each property and class in the ontology as a unit and represent the individuals

as an instantiation of the unit that represents its class.

The units that represent each class have their arity determined by the number of properties defined in the OWL document 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.

Another representation would include in each class unit a list of the names of its individuals, defined in the predicate `individual/1`, where each individual would be represented in a unit named after the individuals name.

Alternate representations problems

The presented representations were tested as possible representations for ontologies in XPTO but were later abandoned as we evolved to the one described in section 4.2.1. Although these were two alternatives we explored, many more exist. The first representation was abandoned due to:

- The possibility of having an arbitrarily large number of arguments in the class units arguments (equal to the number of properties defined in the ontology);
- 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 others may have values for properties that are not present in the representation;

The second approach represents each individual in a separate unit and this could pose a problem as the number of individuals increases, both in terms of representation and querying.

The used representation allows for a more effective individuals search with SPARQL because it only focuses its queries on individuals and not classes. The current representation also allows for a more transparent switch of backend: by changing only the unit individuals one can access the ontology individuals, individuals represented in a database or an external SPARQL agent.

4.2.3 Name analysis

Back to XPTO work-flow, the next process in the loading of ontologies consists in parsing the created term and building a dictionary structure with

all the information needed to generate the units and predicates that will represent the ontology.

The body of the ontology has information about all classes, properties, individuals and relations between these elements. Ontology headers are also stored to be included in the ontology definition unit.

The Symbol Table 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 information 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 now parsed according to the specifications of the OWL language as detailed in [MvH05]. We now present some of the representation or coding 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 4.6. Classes like this are represented internally like any other OWL class and, in order to do this, they are assigned an internal name (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).

AllDifferent

The `owl:AllDifferent` constructor indicates that all the individuals in its list are different from each other and, as stated in section 4.2.1, it 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, we generate `owl:differentFrom` facts relating it to every other individual that comes after it in that list. Since the constructor `owl:differentFrom` is symmetric, this will relate all the individuals between them without generating redun-

```
<owl:allValuesFrom>
  <owl:Class>
    <owl:oneOf rdf:parseType="Collection">
      <owl:Thing rdf:about="#CheninBlancGrape" />
      <owl:Thing rdf:about="#PinotBlancGrape" />
      <owl:Thing rdf:about="#SauvignonBlancGrape" />
    </owl:oneOf>
  </owl:Class>
</owl:allValuesFrom>
```

Figure 4.6: AllValuesFrom example

dant information. For instance, the element in Figure 4.7 will generate the facts represented in Figure 4.8 in the unit individuals.

```
<owl:AllDifferent>
  <owl:distinctMembers rdf:parseType="Collection">
    <vin:WineColor rdf:about="#Red" />
    <vin:WineColor rdf:about="#White" />
    <vin:WineColor rdf:about="#Rose" />
  </owl:distinctMembers>
</owl:AllDifferent>
```

Figure 4.7: AllDifferent example

Document Checker Conformance

W3C defines [CR04] what actions an OWL document checker should do. As a syntax checker, it should receive a document as input and identify it as being Lite, DL, Full or Other

Although XPTO does not currently perform complete conformance tests, this is a subject marked as future work. Nevertheless, it already performs some consistency tests which we now describe:

- A property cannot be a subproperty of a property that is not of the same type e.g., a `DatatypeProperty` cannot be subproperty of an `ObjectProperty` and vice versa;

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

Figure 4.8: AllDifferent representation

- Only constructs allowed by the selected owl variant are used, for example, it is not possible to use `owl:hasValue` in OWL Lite or apply a `owl:InverseFunctionalProperty` to a `datatype property` in OWL DL.

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 [BvHH+05]. Annotations are currently being discarded by XPTO and are marked as future work. One possible manner to represent them would have been to define a predicate `annotation/1` in the unit of the element that the annotation corresponds to (property, class, etc).

Within the ontology headers are the namespaces. Those describe a precise indication of what specific vocabularies are being used in the ontology document. Namespaces provide a means of unambiguously interpreting identifiers and making the rest of the ontology presentation much more readable.

The namespaces of the ontology are being stored by XPTO in the `ontology` unit. 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.

4.2.4 Unit generation and loading

The unit generation process 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 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 later in section 4.3.1, can be added to 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. If the query succeeds the `item/1` predicate will return, by backtrack, the name of the individuals that are valid for the query. Querying property values can be achieved by adding to the context the unit that represents the property (Figure 4.9) or by the inclusion of the unit `property/2` to access a value without knowing the name of the property. An example is shown in Figure 4.10.

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 4.9, the complete context is

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

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

Figure 4.9: Ontology query (direct access)

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

Figure 4.10: unit property definition

shown in Figure 4.11. The `individuals/0` unit is the unit that contains the individuals and property values. The unit `access/0`, partially represented in Figure 4.12, is responsible for accessing the individuals of the ontology or a specific ontology class by instantiating the argument of the `item/1` goal. This is the individual name that will be used by the other units in the context.

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

Figure 4.11: Query context 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

We new present some units which may be used in the queries to retrieve other values or perform additional operations:

```
:- 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 4.12: Unit access

individual/1 Including this unit in the context unifies the argument of the unit with the individual name. Using this unit provides a more explicit query, by asking the individual name and calling the goal `item/0`. It is also possible to query the individual name by using the `item/1` goal. Use of this unit is shown in Figure 4.13.

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

Figure 4.13: Individual example

class/1 If this unit is included in the context it will unify its argument with the class of the matching individual (Figure 4.14). 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' ?
```

Figure 4.14: Class example

property/2 This unit allows to access the properties of the individual without prior knowledge of its name or to query for the property name based on the property value. The first argument is the property name and the second the property value (Figure 4.15).

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

I = 'SelaksIceWine'
P = locatedIn
V = 'NewZealandRegion' ?
```

Figure 4.15: Property example

all/2 Including this unit in the execution context is analogous to using a `findall` in Prolog. The first argument is the element 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, as exemplified in Figure 4.16.

```
| ?- 'Chardonnay' /> individual(I):> all(I, L) :> item.

L = ['BancroftChardonnay',
     'FormanChardonnay',
     'MountEdenVineyardEdnaValleyChardonnay',
     'MountadamChardonnay',
     'PeterMccoyChardonnay']
```

Figure 4.16: All example

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

4.3.2 Native Prolog query representation

To make simple queries easier on Prolog programmers, we created custom predicates that 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 4.17. This allows the user to choose which properties will be present in the generated predicate. The generated Prolog representation is listed in Figure 4.18.

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

Figure 4.17: Predicate definition example

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

Figure 4.18: Generated predicate

This approach is limited because of the fixed arity of the predicates. Some individuals may not have a value for all the properties (an unbound variable for that property will be returned in this case) and other individuals may have properties that are not present in the predicate. It does, however, conform to standard Prolog programming practice, by allowing the usage 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

We now present some use case examples for XPTO. First we compare the expressiveness of XPTO queries with SPARQL [2]. Then is presented a

[2]The SPARQL language is introduced in more detail in 5.1.

scenario in which we combine access to an ontology using the XPTO system and access to data from a database using ISCO [AN06].

4.4.1 SPARQL Query examples

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

Example 1

This first query (Figure 4.19) 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)". The corresponding query in XPTO is shown in Figure 4.20.

In XPTO, the SELECT statement is not used internally, it is implicitly defined by the Prolog variables present in the query. As stated in section 4.2.3, 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 4.19: SPARQL Query example 1

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

Figure 4.20: Query example 1 - internally

Example 2

This query (Figure 4.21) is: "Invert the relation author (from a book to an author) into a relation authored (from an author to a book)."

It 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 aditional Prolog goals. The query that returns the data necessary is shown in Figure 4.22. The use of the `individual/1` unit has the same effect as using the `item/1` goal.

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

Figure 4.21: SPARQL Query example 2

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

Figure 4.22: Query example 2 - internally

Example 3

This query is stated as: "Return the co-author relation between two persons that stand in author relationships with the same book" (Figure 4.23). The correspondent XPTO query is shown in Figure 4.24.

4.4.2 Data Integration: Databases and Ontologies

In this section we will demonstrate, with an example around the Periodic Table, how to write a GNU Prolog/CX program and, using XPTO, to query over two different data sources, namely databases and ontologies.

```
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 4.23: SPARQL Query example 3

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

Figure 4.24: Query example 3 - internally

The Periodic Table

For example purposes, we will use two data sources of information about the periodic table [3]. One will be an ontology [4] that describes the main components of the periodic table like Groups, Blocks and Elements name and the other a database with detailed information about each element. Combining both, we can have access to information such as what are the detailed characteristics of the elements that belong to a particular Group or Period.

Accessing the data

When analysing the definition of a Group in the referred Periodic Table ontology, we can see that each group has, among others, a number, a name and elements. For example, part of group 10 is shown in (Figure 4.25).

Information about the periodic table elements is present in a database defined with ISCO [AN06]. Part of the table `element` definition is illustrated in Figure 4.26.

Having both the referred ontology loaded into our system and the database accessible via ISCO, we can write Prolog programs to reason over both data

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

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

```
<Group rdf:ID="group_10">
    [...]
    <number rdf:datatype="\&xsd;integer">10</number>
    <element rdf:resource="#Ni"/>
    <element rdf:resource="#Pd"/>
    <element rdf:resource="#Pt"/>
    <element rdf:resource="#Uun"/>
```

Figure 4.25: Group 10

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

Figure 4.26: Element Table

sets. Using the Group defined above by the ontology and the elements table defined in the database, we could ask, for example, what is the classification and color of all the elements belonging to the group group_10 as shown in Figure 4.27.

```
| ?-  % 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 4.27: Query example using ontologies and databases

Variables ELEMT and NUM will tie together both data sources and, using the Prolog backtrack mechanism, CLASSF, ELEMT and COLOR will return all the solutions available.

4.5 Experimental Assessment

This section presents the experimental assessment. We begin by comparing the XPTO parser with other XML parsers. Then we present the benchmarks of the representation of ontologies and compare it to other similar systems.

4.5.1 XML Parsers

Next are presented the benchmark results of the parsers we tested. The ontology documents used 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 were tested in a dedicated workstation: a Intel Pentium 4 with hyper-thread running at 3.2Ghz with 1GB of RAM.

Parse times were 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. For instance, the Expat parser [Coo06] is used by the *Mozilla* browser 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 4.28 show the results obtained. The times labeled as `pl-expat` and `pl-libxml2` are those of each parser integrated with Prolog. They do not return anything to Prolog, the difference is that they are called from a Prolog process. These benchmarks are only for time the Prolog overhead.

Table 4.1: Libxml2 and Expat Comparison (seconds)

File (MB)	Expat	pl-expat	libxml2	pl-libxml2
file02 (3,5)	0.04	0.06	0.07	0.08
file03 (1.2)	0.03	0.04	0.07	0.07
file10 (5.5)	0.14	0.16	0.25	0.30
file13 (1.6)	0.04	0.05	0.06	0.07
file17 (24.8)	0.69	0.79	2.02	2.04
file19 (2.6)	0.05	0.06	0.10	0.10
file21 (2.3)	0.05	0.06	0.09	0.10
file22 (14.4)	0.37	0.45	0.80	0.80
file25 (21)	0.55	0.63	0.97	0.97
file27 (32.9)	0.62	0.75	1.61	1.84
file33 (98)	2.68	3.10	4.82	4.83
file34 (4.5)	0.12	0.15	0.25	0.25

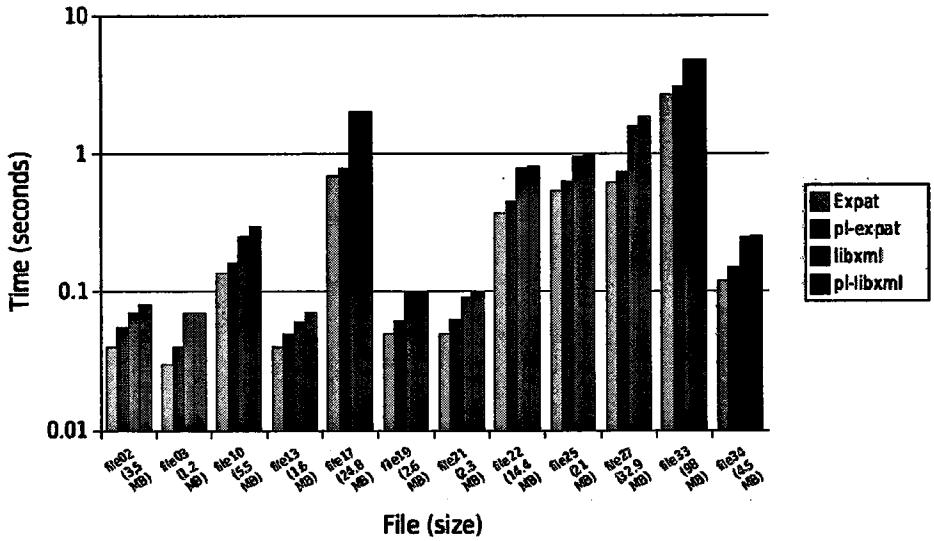

Figure 4.28: Expat Library vs Libxml2

As Table 4.1 illustrates, in both tested cases the Expat library presents better times that Libxml2. As we can see from Table 4.1 and Figure 4.28, the impact on integrating Prolog with Libxml2 is virtually irrelevant. For Expat, 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 XPTO parser (`pl-expat-v2`) uses the Expat library. It builds the structures and terms that represent the XML file and returns the term to Prolog. The implemented parser module was benchmarked against other existing XML parsers. Among the many available XML parsers, we choose the following:

PiLLoW (in GNU-Prolog): Pillow [GH01] is a Web programming library developed at UPM - Technical University of Madrid that provides a way of full 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: W4 [Dam07] is a non-validating parser written in XSB Prolog by Carlos Damásio 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.

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, we used two packages: 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 Prolog system that allows both restricting and extending the language. It supports programming with functions, constraints, objects and features a good base for distributed execution and parallel execution. It has a module that implements the predicates of the PiLLoW package related to HTML/ XML generation and parsing.

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

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, where `pl-expat-v2` is the parser used in XPTO.

Overall, the SWI parser revels the best results, both in terms of parse times and number of files parsed. The XPTO parser, as expected, presently cannot handle the large files, however it presents good results for smaller files (up to 6 MB). The W4 times are high due to the parser validation it performs (in terms of encodings) and represents the whole information in the file. The Ocaml, Pillow and Ciao parsers are not able to parse some files due to not recognizing statements encountered.

Table 4.2: Benchmark results (seconds)

File (MB)	ciao	pl-expat-v2	jena	ocaml	swi	w4	pillow
file02 (3,5)	3.70	0.19	2.08	1.40	0.33	9.96	2.94
file03 (1.2)	0.91	0.89	1.51	-	0.19	3.55	1.15
file10 (5.5)	-	17.3	3.48	6.76	0.82	16.74	-
file13 (1.6)	1.24	0.84	2.37	1.66	0.20	3.25	1.62
file17 (24.8)	-	-	-	-	4.29	84.25	-
file19 (2.6)	1.99	2.65	1.51	2.82	0.36	7.81	2.29
file21 (2.3)	1.67	2.20	1.58	1.66	0.30	6.63	2.05
file22 (14.4)	-	-	23.03	-	2.26	45.34	-
file25 (21)	-	-	10.89	25.20	3.39	63.86	-
file27 (32.9)	-	-	-	39.03	4.75	64.97	-
file33 (98)	-	-	-	127.37	17.00	-	-
file34 (4.5)	-	73.22	3.86	-	0.79	13.43	4.46

Speedups

In addition to the above results we calculated *speedups*. The purpose is to compare the parser results using the times of one of the parsers as base. This enables a better understanding of the results as they are directly compared with a reference.

Table 4.3 shows the obtained results. For the base results we choose the times obtained by Jena since at the time of this writing it has one of the most used parsers among our choices. So, all the times present in Table 4.3 are calculated by dividing the Jena time by the parser time.

Table 4.3: Speedups results (seconds). Jena used as reference

File (MB)	ciao	pl-expat-v2	ocaml	swi	w4	pillow
file02 (3,5)	0.56	10.85	1.49	6.22	0.21	0.71
file03 (1.2)	1.66	1.69	0	7.91	0.43	1.31
file10 (5.5)	0	0.20	0.51	4.22	0.21	0
file13 (1.6)	1.92	2.83	1.43	11.64	0.73	1.47
file19 (2.6)	0.76	0.57	0.53	4.17	0.19	0.66
file21 (2.3)	0.95	0.72	0.95	5.32	0.24	0.77
file22 (14.4)	0	0	0	10.2	0.51	0
file25 (21)	0	0	0.43	3.21	0.17	0
file34 (4.5)	0	0.05	0	4.91	0.29	0.87

Figure 4.29 graphically illustrates the results obtained by our *speedup* cal-

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

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

culations. Analyzing the values we can conclude that the parser we choose as reference 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 we can look at Table 4.4 where these are represented as percentages. Looking at our parser results we can conclude that it has better parsing times (in relation to Jena) for 3 files, worse times for 4 files and that it can not parse two files that Jena can (the two bigger ones). The largest difference is in `file02`, where XPTO presents a measured time 985% better than Jena.

4.5.2 Ontology representation benchmarks

Next the benchmarks of the complete representation of the ontology are presented. For XPTO this includes the parse, name 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 briefly described next:

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].

Pellet [SP04] is a open source reasoner for the OWL DL ontology language

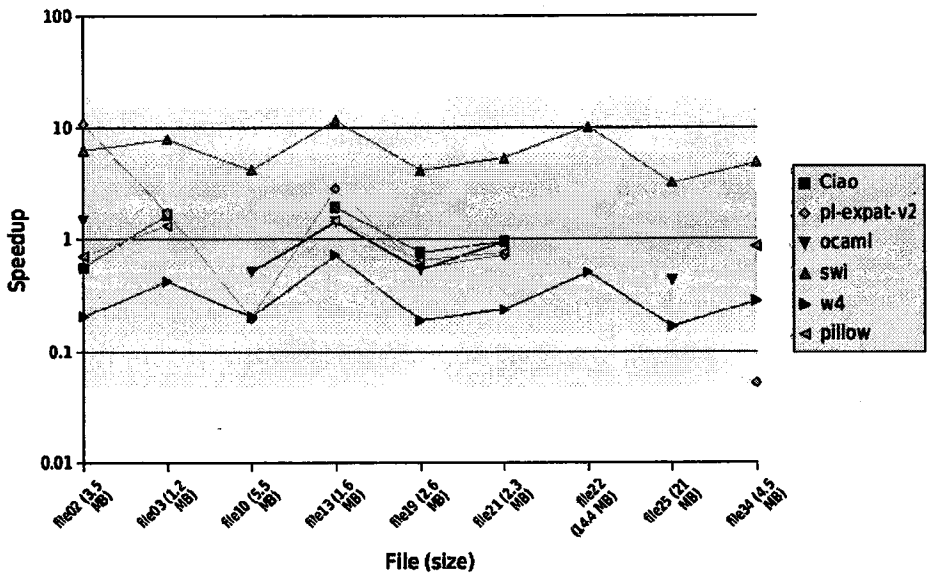

Figure 4.29: Speedup graph

developed at the Mindswap Lab [5] of the University of Maryland. Pellet contains a query engine which supports answering queries formulated using SPARQL and supports reasoning with multiple ontologies.

Pellet also implements a species verification when parsing the ontology, but the times were measured with this feature disabled. Thea represents the ontology as predicates stored in its Prolog knowledge base. The representation of the ontology adopted in XPTO is described in section 4.2.1.

Table 4.5 contains the times calculated for all the systems. These times were measured using the same method as described in section 4.5.1.

The values present in Table 4.5 allow us 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 XPTO is, on average, 97.5% times slower than the Pellet system, as shown in the *speedups* Table 4.6. The XPTO system is further timed next in section 4.5.3 where explinations for the slowdown are given.

[5] http://mindswap.org/

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

File (MB)	Thea	XPTO	Pellet
file35 (2.3)	21.22	206.33	4.39
file36 (1.2)	6.96	98.78	2.61
file37 (2.2)	105.15	204.91	5.57
file38 (1.2)	4.66	96.4	2.50

Table 4.6: Performance gain of representing the ontologies

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

4.5.3 XPTO time analysis

In this section we analyze the time it takes for XPTO to parse and build the representation of each file. These times are measured using the `statistics/2` predicate of GNU Prolog, using the `real_time` statistics key.[6]

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 39);

build: Time to build the dictionary;

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

compile: 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, we can 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.

[6]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 (MB)	parse	build	print	compile	load
file35 (2.3)	0.01	0.03	0.02	0.79	0.15
file36 (1.2)	0	0.04	0.02	0.86	0.07
file37 (2.2)	0.01	0.04	0.01	0.80	0.14
file38 (1.2)	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 a prototype system for representing ontologies. The achieved implementation covers our initial objectives: we obtained a functional system that can reason over ontologies from a contextual programming point of view. The presented and discussed results are not optimal and showed us where we can improve the system's performance.

Currently there may be only one ontology loaded at a particular time. This represents one of the major shortcomings of the current state of the work. The objective is to be able to load an arbitrary number of ontologies and perform mixed queries over these. In order to achieve this some changes will have to be made to the representation. These include changing the name of the units (and unit files) to avoid name clashes, and altering the query method described in section 4.3 to take into account the several ontologies. The unit `individuals` will also have to be changed to guarantee the separation of individuals of the loaded ontologies, possibly becoming several units, one for each loaded ontology.

The fully support of a well defined OWL sub-language is also work that has to be accomplished in future work. The parser is simple and not very robust. It should be improved in order to be more efficient and to handle larger ontologies.

Chapter 5

SPARQL Back-end for Contextual Logic Agents

XPTO is an ontology mapping engine that aims for the creation of a logic layer over ontologies. Access to this system provides a way for reasoning over ontologies by means of Logic Programming. However, the work done with XPTO is meant to be viewed as the foundation of a hub system in which different entry and exit plug-ins for data access can be used. For example, the ontology representation and access provided by XPTO can be used by a SPARQL Front-end that receives a SPARQL query about an ontology loaded in the system and returns the solution.

This chapter presents a back-end for XPTO that aims to transparently merge the reasoning of the system's internal knowledge base with external ontologies available from third parties, by means of the SPARQL query language. To achieve this, we developed a system that provides functions for communicating with Web SPARQL agents for ontology querying purposes. It provides the system with the ability to pass a SPARQL query to an arbitrary SPARQL Web agent and get the solution, encapsulating the results as bindings for logic variables.

This SPARQL back-end grants XPTO capabilities for writing GNU Prolog/CX programs to reason simultaneously over local and external ontologies. But equally important, this back end implementation provides roots for, as future work, implementing access and reasoning capabilities for as many different data sources as possible, as shown in Figure 5.1.

The remainder of this chapter is organized as follows: first, sections 5.1 and 5.2 introduces SPARQL and its query protocol. Then, section 5.3 introduces and describes the implementation of the XPTO SPARQL back-end.

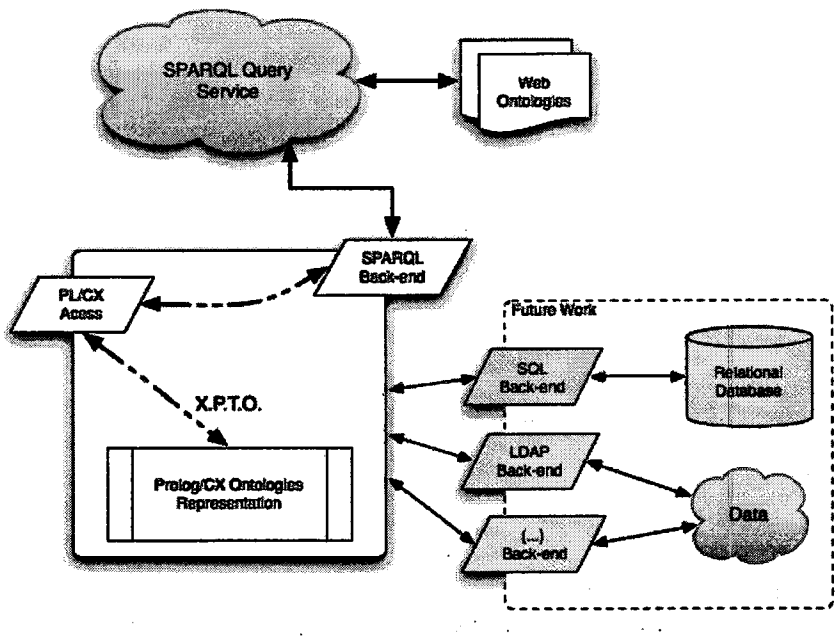

Figure 5.1: System architecture

Finally, in section 5.4 we present some results and draw conclusions.

5.1 Querying with SPARQL

SPARQL is a Semantic Web query language developed by the W3C working group. At the time of this writing, one of the most important open research issues in the Semantic Web is the lack of a query language standard specification that can access data described by ontologies. There exists a variety of Semantic Web query languages projects [TF06], ranging from pure selection languages with limited expressivity to general purpose languages supporting different data representation formats and complex queries. However, none of them is currently referenced as standard. Among all the possibilities, our choice was to use SPARQL [PS06]. SPARQL is a query language that is suitable for both local and remote use and access. For remote use, the SPARQL Protocol for RDF [Cla06] has been designed and is introduced in more detail in section 5.2.

SPARQL was designed to meet the requirements and design goals described in the RDF Data Access Use Cases [Cla07] document. It provides

capabilities for:

- Extracting information in the form of URIs, blank nodes, plain and typed literals;
- Extracting RDF sub-graphs;
- Constructing new RDF graphs based on information present in the queried graphs;

As the first item of the previous list states, information can take the form of URIs - Uniform Resource Identifier (or IRIs[1] - *International Resource Identifiers*, which are a generalization of URIs and URLs). Blank nodes are variables that have no name but can be referenced within a graph pattern. In order to better understand RDF concepts, let's look at Table 5.1, which draws a comparison between XML and RDF querying:

Table 5.1: Querying XML compared to querying RDF

Concept	XML	RDF
Model	Document or Tree	Set of Triples = RDF graph
Atomic Units	Elements, Attributes, Text	Triples, URIs, Blank Nodes, Text
Identifiers	Element/Attribute Names, IDs	URIs
Described by	DTDs, W3C XML Schema	RDF Schema

The main difference resides in the data model. XML is a structured document, usually in a tree form. RDF is a graph data model, a set of triples in the form (*Subject, Predicate, Object*).

As a query language, SPARQL is data oriented, which means there is no query inference in the query language itself, it only queries the information held in the models. It provides Semantic Web users with a query language in much the same fashion as SQL provides relational database users with a query language: It does not do more than take the description of what is wanted, in the form of a query, returning that information in the form of a set of bindings or an RDF graph.

[1]Internationalized Resource Identifiers, described by RFC3987 at http://www.ietf.org/rfc/rfc3987.txt

5.1.1 How to write SPARQL queries

An RDF graph is a set of triples in the form (*Subject, Predicate, Object*). The SPARQL queries are based on matching graph patterns, where the simplest graph pattern is similar to an RDF pattern, but with the possibility of a variable instead of an RDF term in the subject, predicate or object positions. The combination of triple patterns results in a basic graph pattern.

Let us study a first query example. Given the RDF graph in Figure 5.2, we want to write a SPARQL query to find who is the director of a movie from the information in the given RDF graph.

```
1 <http://example.org/movie/movie1>
2 <http://xpto.org/persons/director> "David Fincher".
```

Figure 5.2: RDF graph example

The query (Figure 5.3) has two parts, a `SELECT` and a `WHERE` clause. The first is responsible for identifying the variables that will appear in the query results and the second encapsulates the triple patterns. The result will match the variable *director* with *"David Fincher"*.

```
1 SELECT ?director
2 WHERE
3 {
4 <http://example.org/movie/movie1>
5 <http://xpto.org/persons/director> ?director .
6 }
```

Figure 5.3: SPARQL query example

SPARQL Syntax

The terms between <> are URIs. SPARQL provides two different kinds of abbreviations for URIs:

1. `PREFIX`. This keyword associates a prefix label with an URI. A prefixed name is a prefix label and a local part separated by a colon ":". A prefixed name is mapped to an URI by concatenating the URI associated

with the prefix and the local part. The prefix label or the local part may be empty.

2. `BASE`. This keyword defines the base URI used to resolve relative URIs.

The general syntax for literals is a string with either an optional language tag (introduced by @) or an optional data type URI or prefixed name. A string can be written enclosed in quotes; either double `""` or single `''`. For convenience, integers and decimals can be written directly without quotes but using explicit URI data types: `xsd:integer`, `xsd:decimal` and `xsd:double`. Boolean values can also be written without quotes, using `xsd:boolean`.

Query variables have global scope. The use of a given variable name anywhere in a query always identifies the same variable. Although variables are prefixed by either `"?"` or `"$"`, both characters are not part of the variable name. For example, `$name` and `?name` identify the same variable in a query.

As shown in Table 5.1, an RDF node can be an URI, a literal or a blank node. A blank node is a node that is not a URI reference or a literal. In the RDF Abstract Syntax specification [KC07], a blank node is just a unique node that can be used in one or more RDF statements, but has no intrinsic name. In SPARQL, blank nodes in graph patterns act as non-distinguished variables, not as references to specific blank nodes in the data being queried.

Graph patterns

SPARQL is all about graph pattern matching, where complex graph patterns can be formed by combining smaller patterns in various ways:

- **Basic**, where a set of triple patterns must match;
- **Group**, where a set of graph patterns must all match;
- **Optional**, where additional patterns may extend the solution;
- **Alternative**, where two or more possible patterns are tried;
- **Patterns on Named Graphs**, where patterns are matched against named graphs;

Figure 5.4 illustrates a query with a group graph pattern (delimited with {}) of one basic graph pattern. Using the keyword `FILTER`, a constraint can

```
PREFIX foaf:    <http://xpto.org/foaf/>
SELECT ?name ?mbox
WHERE  {
          ?x foaf:name ?name .
          ?x foaf:mbox ?mbox .
          FILTER regex(?name, "Mr. French")
       }
```

Figure 5.4: SPARQL Group graph pattern

be applied to the query in order to restrict the solutions over the whole group in which the filter appears

In basic graph patterns, the entire query pattern must match in order to give a solution. However, it is useful, in certain scenarios, to not reject the solution because some part of the query pattern does not match. `OPTIONAL` matching provides this facility which means that if the optional part does not match, it creates no bindings but does not eliminate the solution. Figure 5.5 illustrates an example of a SPARQL query that uses a constraint in an optional graph pattern.

```
PREFIX  dc:  <http://xpto.org/dc/objects/>
PREFIX  ns:  <http://xpto.org/ns#>
SELECT  ?title ?length
WHERE   { ?x dc:title ?title .
          OPTIONAL { ?x ns:length ?length .
                     FILTER (?length < 120) }
        }
```

Figure 5.5: SPARQL optional group graph pattern

Supposing that in the RDF graph we have two movies, one with length equal to 90 minutes and other with 130, the solutions would be the ones shown in Table 5.2.

Matching alternative graph patterns can be done using the `UNION` keyword between graph patterns. If more than one of the alternatives matches, all the possible pattern solutions are returned.

Table 5.2: Solutions to Figure 5.5 query

title	length
"The Departed"	
"Reservoir Dogs"	99

In the previous examples, all the queries have been shown executed against a single graph, the default graph of an RDF data-set which acts as the active graph. An RDF data-set comprises one default graph which does not have a name and zero or more named graphs, where each named graph is identified by an URI. The graph that is used for matching a basic graph pattern is the active graph. The `GRAPH` keyword is used to make the active graph one of all of the named graphs in the data-set for part of the query.

A SPARQL query may specify the data-set using the `FROM` and `FROM NAMED` clauses. They both indicate that the data-set should include graphs that are obtained from representations of the resources identified by the given URIs. The data-set resulting from these clauses is:

- A default graph consisting of the RDF merge of the graphs referred to in the FROM clauses;
- A set of (URI, graph) pairs, one from each FROM NAMED clause;

If there is no FROM clause, i.e, a default graph, but there is one or more FROM NAMED clauses, then the data-set includes an empty graph for the default graph. The query in Figure 5.6 matches the graph pattern against each of the named graphs in the data-set and forms solutions which have the `src` variable bound to URIs of the graph being matched. The graph pattern is matched with the active graph being each one of the two named graphs in the data-set.

The query result gives the name of the graphs where the information was found and the value for jules's nickname, as showed in Table 5.3.

Table 5.3: Solutions to the query in Figure 5.6

src	name
http://example.org/foaf/jules	"Jules Winnfield"
http://example.org/foaf/brett	"Pit"

Query patterns generate an unordered collection of solutions. These solutions are then treated as a sequence where initially there is no specific order.

```
PREFIX person: <http://xpto.org/persons/>

SELECT ?src ?name
FROM NAMED <http://xpto.org/persons/jules>
FROM NAMED <http://xpto.org/persons/brett>
WHERE
  {
    GRAPH ?src
    { ?x person:mbox <mailto:jules@work.example> .
      ?x person:nick ?name
    }
  }
```

Figure 5.6: Named graphs

However, SPARQL has sequence modifiers constructors that can then be applied to create a different sequence. The following list comprises the solution sequence modifiers available:

- **Order**: order the solutions;
- **Projection**: choose certain variables;
- **Distinct**: ensure solutions in the sequence are unique;
- **Reduced**: permit elimination of some non-unique solutions;
- **Offset**: control where the solutions start from in the overall sequence of solutions;
- **Limit**: restrict the number of solutions;

Query Forms

SPARQL has four query forms. Each one of them uses the solutions from pattern matching to form result sets. The query forms are:

SELECT: returns all, or a subset of, the variables bound in a query pattern match. `SELECT *` is used to select all the variables in the query. The `SELECT` form is currently the only implemented form by the back end of XPTO, whereas the other three are marked as future work.

CONSTRUCT: returns an RDF graph constructed by substituting variables in a set of triple templates.

DESCRIBE: returns an RDF graph that describes the resources found.

ASK: returns a boolean indicating whether a query pattern matches or not.

Among those query return forms, two of them return variable bindings and the other two return RDF graphs. For the first group (`SELECT` and `ASK`), results can be thought of as a table with one row per query solution, where some cells may be empty because a variable is not bound in that particular solution (see example in Figure 5.2). These result sets can be serialized into either XML or an RDF graph; for the XML serialization, an XML format is described in the W3C Candidate Recommendation *SPARQL Query Results XML Format* document [BB06]. This document describes an XML format for the variable binding and boolean result formats provided by SPARQL . Let us exemplify this by first looking at the RDF triples present in Figure 5.7 (expressed in Turtle syntax [Dav07]):

```
1 @prefix  person:  <http://xpto.org/persons/> .
2 _:a    person:name   "Beatrix Kiddo" .
3 _:a    person:knows  _:b .
4 _:a    person:knows  _:c .
5 _:b    person:name   "Budd" .
6 _:c    person:name   "O-Ren Ishii" .
7 _:c    person:nick   "Cotton Mouth" .
```

Figure 5.7: Turtle RDF graph example

Given this RDF graph, Figure 5.8 shows a query that selects the names of persons that are known by persons present in the triples and their nicknames if any.

This will result in the set composed by the names *Budd* and *O-Ren Ishii*, and the nickname of *Cotton Mouth*. The XML format is presented in Figure 5.9. The `head` tag encapsulates the variables that are to be return and the `results` tag encapsulates all the bindings for those variables.

This section presented an introduction and an overview of the SPARQL query language. Most of the aspects focused are the ones that have importance and are relevant in the work done with the back end presented next in this chapter. For further details consult the three SPARQL specifications [PS06, Cla06, BB06].

```
1 PREFIX persons: <http://xpto.org/persons/>
2 SELECT ?name ?nick
3 WHERE
4   { ?x persons:knows ?y.
5     ?y persons:name ?name .
6     OPTIONAL { ?y persons:nick ?nick }
7   }
```

Figure 5.8: Nickname SPARQL query example

5.2 Querying an external SPARQL agent

As a data access language for the Semantic Web, SPARQL is suitable for both local and remote use. For remote use, the W3C group is working on a SPARQL protocol for Web agents communication [Cla06]. This document describes means of conveying SPARQL queries to SPARQL query services and how the query results are returned to the requesting entity, where bindings like HTTP and SOAP [2] have been introduced to achieve connectivity.

5.2.1 SPARQL Protocol

SPARQL protocol uses the Web Services Description Language (WSDL) [RC07] in order to describe a way to elaborate SPARQL queries to a SPARQL query processing service and returning the query results. This protocol, developed by the W3C RDF Data Access Working Group (DAWG) as part of the Semantic Web activity, is described in two ways: as an independent abstract interface and as HTTP and SOAP bindings to this interface.

`SparqlQuery` is the only protocols interface and it contains one operation, `query`, which is used to specify a SPARQL query string. The query operation is described as an *In-Out* message exchange pattern, which means it consists of exactly two messages, where the first is the *In* - query request and the second is the *Out* - query result.

The content of an *In* message is composed of two further parts: one SPARQL query string and zero or one RDF data-set descriptions. The query

[2] Simple Object Access Protocol is a protocol for exchanging XML based messages in a decentralized, distributed environment. Consult the appropriate W3C specification [FN07] for detailed information.

```
<?xml version="1.0"?>
<sparql xmlns="http://www.w3.org/2005/sparql-results#">
  <head>
    <variable name="name"/>
    <variable name="nick"/>
  </head>
  <results>
    <result>
      <binding name="name">
        <literal>Budd</literal>
      </binding>
   </result>
    <result>
      <binding name="name">
        <literal>O-Ren Ishii</literal>
      </binding>
      <binding name="nick">
        <literal>Cotton Mouth</literal>
      </binding>
    </result>
  </results>
</sparql>
```

Figure 5.9: XML response format for query in Figure 5.8

string is defined as a sequence of characters in the language defined by the SPARQL grammar, starting with the query production. The RDF dataset description is formed by zero or one default RDF graphs composed by the merge of zero or more default or named graphs (`FROM` and `FROM NAMED` keywords in SPARQL, respectively). The *Out* message is an instance of an XML Schema type called `query-result` composed of either:

1. A SPARQL *results* document in response to a `SELECT` or a `ASK` query form;

2. An RDF graph in response to a `DESCRIBE` or `CONSTRUCT` query form;

If an operation fault rises, the `query` operation contained in the `SparqlQuery` interface may return, in place of the *Out* message, either the predefined

`MalformedQuery` or `QueryRequestRefused` message, both also defined in XML Schema.

HTTP Bindings

The `SparqlQuery` interface described above is an abstract operation, which means it requires specific protocol bindings in order to become an invocable and usable operation. Next we describe the `HTTP` binding to the SPARQL protocol as it is the one used by the implemented XPTO back-end.

There exists two `HTTP` bindings: **`queryHttpGet`** and **`queryHttpPost`**. The `query` operation binding uses the HTTP `GET` with the following serialization types constraints:

- Input: application/x-www-form-urlenconded and application/xml
- Output: application/sparql-results+xml and application/rdf+xml

Let us see a simple example. Figure 5.10 shows a query that is sent to some SPARQL query service located, say, at *http://xpto.org/service/sparql/*. The operation is illustrated in the `HTTP` trace presented in Figure 5.11.

```
1 PREFIX dc: <http://xpto.org/movies/>
2 SELECT ?movie ?director
3 WHERE { ?movie dc:directed ?director }
```

Figure 5.10: SPARQL simple Query example

```
1 GET /sparql/?query=PREFIX+dc:+&lt;
2 http://xpto.org/movies/&gt;%13SELECT+?movie+?director%13WHERE+
3 {+?movie+dc:directed+?director+}
4 Host: xpto.org/service/
5 User-agent: sparql-client/0.1
```

Figure 5.11: SPARQL external query operation

In the `GET` request there is an URL encoded SPARQL query and the location of an `HTTP` server. Note that the query is encoded: spaces are

replaced by the plus symbol *"+"* and new lines are replaced by *"%13"*, which is a hexadecimal value of the new line character number. The result is presented in Figure 5.12.

```
HTTP/1.1 200 OK
Date: Fri, 010 Apr 2007 15:45:32 GMT
Server: Apache/1.3.29 (Unix) PHP/4.3.4 DAV/1.0.3
Connection: close
Content-Type: application/sparql-results+xml

<?xml version="1.0"?>
<sparqlxmlns="http://www.w3.org/2005/sparql-results#">

 <head>
   <variable name="movie"/>
   <variable name="director"/>
 </head>
 <results distinct="false" ordered="false">
...
 </results>
</sparql>
```

Figure 5.12: SPARQL external query result

```
GET /sparql/?query=<EncodedQuery>&
default-graph-uri=http://www.other.xpto2.org/movies2 HTTP/1.1
Host: www.other.xpto2.org
User-agent: my-sparql-client/0.1
```

Figure 5.13: SPARQL external HTTP trace example

Note that in this example the RDF data-set is not specified in the query nor in the protocol. The RDF data-set can be specified in the query or directly in the protocol or in both query string and in the protocol. In the case where it is specified in both sides, the specification indicates that the protocol must be the RDF data-set consumed by the query operation. Figure 5.13 shows an HTTP trace where the RDF data-set to be used is passed in

the protocol, identified by the value of the `default-graph-uri` parameter. Anther possible scenario is if the Web service only takes queries against a specific predefined data-set. In this case, there is no need for an explicit the data-set via query or protocol.

5.3 Back-End Processor

We now describe a SPARQL back-end for XPTO that is capable of communicating with SPARQL Web agents. This enables writing GNU Prolog/CX programs to reason simultaneously over local and external ontologies.

5.3.1 Architecture

The back-end engine provides XPTO with a means for querying external Semantic Web services in SPARQL. Although it can be viewed as a single independent component, the purpose is to integrate it in a manner that it will allow the XPTO-using programmer to query external and internal ontologies using the same query syntax and declarative context mechanics as the internal system. This will allow to transparently query internal and external ontologies and merge their results in the same program. Figure 5.14 illustrates the architecture of XPTO with the integration of the SPARQL back-end component.

To achieve this level of functionality, we developed a GNU Prolog/CX to SPARQL engine that satisfies the following requirements:

- Translate a partially bound GNU Prolog/CX goal into SPARQL;
- Send the SPARQL query to the specified Semantic Web SPARQL service;
- Fetch the XML result file, parse it and return the solutions as Prolog variable bindings using the GNU Prolog/CX backtrack mechanism to iterate over sets of answers;

Although the queries are meant to have the same syntax as the ones executed by XPTO, there is additional information that is needed to be provided to the external agent if the SPARQL protocol [Cla06] is to be used. This includes, among others, the `url` of the service, the data format of the response and, possibly, an ontology URI. The latter means that external agents such as the `XML Armyknife` Semantic Web service [Dod06] that is used throughout this section to illustrate the back-end functionality may have capabilities

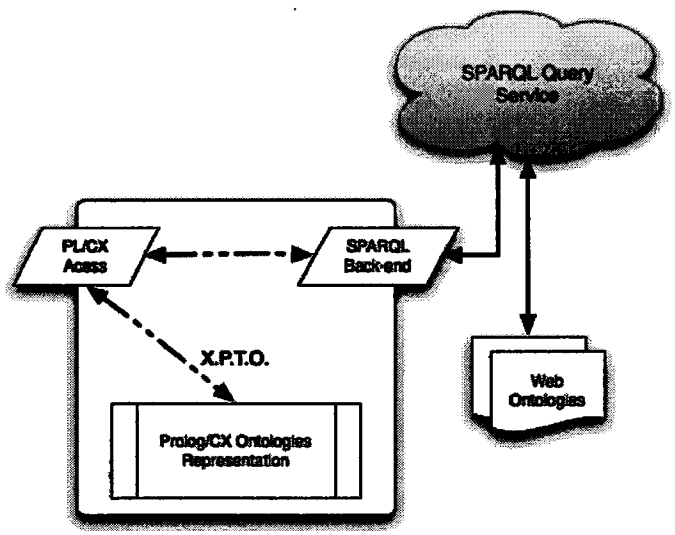

Figure 5.14: System architecture

for querying ontologies from any given Internet location. The response format can vary from different types like simple HTML for Internet browsing purposes, or the SPARQL Query Results XML Format [BB06] for agents like ours.

5.3.2 Prolog/CX to SPARQL mapping

SPARQL is a recent language and is still undergoing an evolution process. It has many different constructs, forms and capabilities. The XPTO back-end that we present here is currently functional, which means it can successfully translate a GNU Prolog/CX query into SPARQL, communicate with outside agents, get the response and return the solutions as bindings to Prolog variables. At the time of this writing there are limitations on the queries which can be generated:

- Queries must form basics graph patterns (simple triple patterns);
- Only produce the `SELECT` query form;
- Data sets defined by the clauses `FROM` and `FROM NAMED` must be defined as facts;

- Abbreviations must be defined as facts;

A SPARQL query in the back-end environment is a GNU Prolog/CX context execution similar to the ones defined by the XPTO main mapping engine. The query is always composed of three parts:

1. One URI of the external Semantic Web service;
2. One or more property restrictions;
3. An execution predicate which refers individuals;

Figure 5.15 illustrates a definition of a back-end query and its three components.

```
1 QUERY := sparql(URI) /> P1 ... Pn :> ITEM
2
3       URI  := URL
4       P    := property(VALUE) or where(PROP, VALUE)
5       ITEM := item(INDIVIDUAL)
```

Figure 5.15: Back-End Query Definition

As the above figure shows, on the left side of the `'/>'` operator are the connection parameters and on the right the query properties and individuals.

The main syntactic difference between the two types of queries (local and external) resides on the left side of the XPTO operator `'/>'`. There are two query situations for local queries and one for external:

1. A class name for local reasoning over objects from that class;
2. Operator by itself for direct *SPARQL-like* local reasoning, i.e, query variables on properties and/or individuals;
3. A compound term with functor `sparql/1` which identifies the target external SPARQL agent;

Note, however, that parameters other than the URI of the external Web Service may be needed for the external query execution. As discussed in section 5.2.1 and defined in the SPARQL Protocol for RDF [Cla06], two more

parameters can be specified: a response file format and the location of the ontology which the system wants to query. The latter is used with Web Services attached not only to one ontology but capable of querying, in SPARQL, any other ontology in the Web, given its location. Each Web Service adopts one response format parameter, although the sender can specify in the query what format the response should take. These parameters must be present in the program knowledge base as Prolog facts or made available in the execution context.

The context execution that composes the query is necessarily different from the queries that reason over local ontologies, as there is no connection between what is included, as properties, in the context execution and whatever data is loaded into the main engine of XPTO, i.e, the properties in a back-end query are not associated with existing predicates that are present in the query context. This means that although the query has the form of a GNU Prolog/CX context execution with the predicate `item/1` as goal, its execution is handled differently. The goal is not executed by any unit that appears in the context on the right side of the query, mostly because what is written as properties of the query will not exist as explicit units in our system and thus, in order for a query context to be formed, the programmer must have prior knowledge of the structure of the ontology being queried. If no properties names or individuals are known, it is possible to query for what properties a certain individual has, as detailed next in this section.

On the left side of the main operator `'/>'` the external agent is specified and on the right side, the goals and query restrictions. The right side of the operator encodes the query that must be mapped to SPARQL. One way to do this is to translate that information into RDF triples, much in the same way a database is translated into triples, i.e, for each of the n stated properties about an individual, the back-end must translate it to `(n-1)` triples. The triples are extracted by the union of each **property** term of the right side and the **item** term, which represents the subject of the triple. Figure 5.16 shows an example.

The query in Figure 5.16 asks for individuals that have the properties **property1** and **property2** and what their values are. All unbound Prolog variables represent variables in the triples. If more than one solution is available for the query, all the results are retrieved using the Prolog backtracking mechanism. To state a value in the query and therefore apply a restriction to the solution, a Prolog atomic value can be used to bond a Prolog variable. In the previous example, the substitution of the variables **V1** and **IND** for

```
The query:
sparql('uri.org') />  property1(V1) :>
                      property2(V2) :> item(IND).

Translates into the RDF triples:
(?IND, property1, ?V1)
(?IND, property2, ?V2)
```

Figure 5.16: Back-End triples generation, Example 1

valueX and *ind1* respectively translates to the triples in Figure 5.17.

```
(ind1, property1, valueX)
(ind1, property2, ?V2)
```

Figure 5.17: Generated triples

The query scheme explained so far only is useful if the names of the properties of the ontology are known. To ask for a property name, i.e, to generate an RDF triple where the property position is a variable, the back-end unit `where/2` should be use. Figure 5.18 illustrates an example which queries for all the properties and their values for the individual named *individualA*.

```
The Query:
sparql('uri.org') />  where(PROP, VAL) :> item('individualA').

Translates into the RDF triples:
('individualA', ?PROP, ?VAL)
```

Figure 5.18: Back-End triples generation, where clause Example

Note that this clause can also be used like a single ask property, by grounding the first argument to a Prolog atom named with some property that describes the individual.

The triples generation process divides the GNU Prolog/CX query information into two parts: the variables, i.e, what is asked and the triple sets. This will result in a direct and transparent translation to SPARQL, where the variables in the query will be the SELECT SPARQL clause arguments and the triples will form the sets in the WHERE SPARQL clause. For example, the query presented in Figure 5.18 will be translated into the SPARQL query presented in Figure 5.19

```
1 SELECT ?prop, ?val
2 WHERE
3 {
4 'individualA' ?prop ?val.
5 }
```

Figure 5.19: GNU Prolog/CX to SPARQL Example 1

The GNU Prolog/CX to SPARQL translation scheme presented in this section covers simple SPARQL clauses and forms. Other SPARQL query forms like CONSTRUCT or DESCRIBE or query patterns like ORDER or OPTIONAL are not implemented or necessary at this point. The generation of SPARQL which satisfies the Prolog operational semantics is the goal of the present work.

5.3.3 Web Agents Communication

After the SPARQL query is constructed, a communication process must be carried out between the back-end and the Semantic Web SPARQL service that is to be used. The back-end implements a simple connection model divided into the following steps:

1. Establish connection;
2. Send query;
3. Receive the response;
4. Close connection;

Before the query it is sent, some transformation work must be done, i.e, the query must be encoded accordingly to the SPARQL Protocol for RDF

[Cla06] (see section 5.2.1). This includes appending optional information to the query like the *prefix*, the default URI or the response file *format*.

After encoding the query, the back-end starts the communication process with the external agent. This represents the *Establish connection* step in the above sequence and includes the validation of the Web service:

- Open the communications;
- Verify whether the external host is up and ready for communication;

After the connection is established, the query is then sent. If everything went well, the external agent response is received and the connection is closed. This represents the remaining steps in the back-end query execution list. Figure 5.3.1 presents the communication process.

Algorithm 5.3.1: EXTERNAL COMMUNICATION PROCESS(*query*)

```
if (connectionsuccessful)
  then { socket ← query
         repeat
           result ← read(socket)
         until
         socketClosed;
         return (result)

  else { error ← "No_hosname_response."
         return (error)
```

After receiving the response and closing the connection, the process returns to the Prolog side. The XML format that represents the solutions to the query follows the specification described in the SPARQL Query Results XML Format [BB06], described in section 5.2.1. The XML which contains all the existing solutions for the query is then parsed. Figure 5.9 in subsection 5.2.1 shows an example of a XML response returned by an external agent. Finally, the back-end will provide each logic solution to the query present in the XML, one at a time if more than one are available.

5.3.4 Examples and Query solutions

To better illustrate how the back-end operates, let us focus on a real example. The SPARQL service used is hosted at *xmlarmyknife.org* [Dod06] and

is called *XAK* [3]. The *XAK* SPARQL query service implements the SPARQL Protocol for RDF [Cla06] and it provides a SPARQL query engine for RDF data available on the Internet. The *XAK* query engine also extends the standard protocol to provide support for multiple output formats.

The following are the most relevant features that characterize *XAK*:

Base URL: `http://xmlarmyknife.org/api/rdf/sparql/query;`

Requested Methods Supported: `GET` and `POST`. A `GET` of the Base URL without any parameters will return an `HTML` form suitable for experimenting with the query service. `POST` is almost equivalent to `GET` and should be used in exceptional cases. The SPARQL Protocol notes that *"[GET] should be used except in cases where the URL-encoded query exceeds practicable limits;"*

Request Parameters: The request parameters supported by this service, with the exception of *format*, are specified in the SPARQL Protocol for RDF [Cla06]. Table 5.4 summarises their use;

Table 5.4: *XAK* request parameters

Parameter	Notes
`query`	URL encoded SPARQL query
`query-uri`	URL from which query can be fetched. Extension to the SPARQL protocol
`default-graph-uri`	Absolute URL of RDF data source(s) to populate the background graph
`named-graph-uri`	Absolute URL of RDF data source(s) to be used as named graphs
`format`	Format for results. Extension to the SPARQL protocol. Values depend on type of query.
`xslt-uri`	Absolute URL of XSLT stylesheet to apply to SELECT query results (ONLY). Extension to the SPARQL protocol.

Response Codes: The following HTTP response codes will be returned by this service:

- 200 - successful query;

[3]The XML Army Knife is a project by Leigh Dodds. For more information visit his Web page at http://www.ldodds.com/

- 400 - malformed query;
- 500 - error processing query or fetching data;

Response Format: By default responses to SELECT queries from this service will conform to the SPARQL Query Results XML Format [BB06]. As specified in that document, the *MIME* type will be *application/sparql-results+xml;*

Additional Response Formats (SELECT): The format parameter can be used to select one of the alternate output formats present in Table 5.5;

Table 5.5: Additional *XAK* response formats

Format Value	*MIME* type
html	text/html
json	application/sparql-results+json
javascript	application/javascript

The first format is an `html` document containing query summary and tabular results. The second format is a `json` [4] serialization of results and the `javascript` format generates an `html` table with a SPARQL class CSS style associated with it.

XAK supports most of the important specifications stated in the SPARQL Protocol for RDF [Cla06], which makes it a reasonable test case for using with the back-end. The description does not mention any tie to a specific ontology and states the support of the parameters *default-graph-uri* and *named-graph-uri*, which means an external ontology must be used for querying purposes.

The *Wine* OWL DL ontology [W3C06] is a sample ontology used in the OWL specification documents and will serve as the use case ontology in this section. The *Wine* ontology defines classes, properties and individuals about different kinds of wines and, with SPARQL, it is possible to query for RDF triples about the information that exists in the ontology. Among others, the `IceWine` class defines two properties: `hasBody` and `hasColor` (Figures 5.20 and 5.21 respectively).

These two properties state that an *IceWine* individual should have, among others, values for the *hasBody* and *hasColor* properties. The first must be one of two kinds: *Medium* or *Full* and the latter should be *White*. The

[4] For more information about the JSON serialization please visit http://www.json.org/

```
<rdfs:subClassOf>
  <owl:Restriction>
    <owl:onProperty rdf:resource="#hasBody" />
    <owl:allValuesFrom>
      <owl:Class>
        <owl:oneOf rdf:parseType="Collection">
          <owl:Thing rdf:about="#Medium" />
          <owl:Thing rdf:about="#Full" />
        </owl:oneOf>
      </owl:Class>
    </owl:allValuesFrom>
  </owl:Restriction>
</rdfs:subClassOf>
```

Figure 5.20: *hasBody* IceWine property

```
<owl:intersectionOf rdf:parseType="Collection">
   <owl:Class rdf:about="#LateHarvest" />
   <owl:Class rdf:about="#DessertWine" />
   <owl:Restriction>
     <owl:onProperty rdf:resource="#hasColor" />
     <owl:hasValue rdf:resource="#White" />
   </owl:Restriction>
</owl:intersectionOf>
```

Figure 5.21: *hasColor* IceWine property

first property is defined as a *subClass* restriction for the *IceWine* individuals where each one of them must have a value for the property *hasBody*. In the other hand, the *hasColor* property is defined first as an individual intersection of two classes (*LateHarvest* and *DessertWine*) and second as a value property restriction that they must have, which is the *White* color.

Figure 5.22 shows an example of a back-end query that asks *XAK* to search the *Wine* ontology for all the individuals that have both of these properties.

The GNU Prolog/CX query in Figure 5.22 has no ground Prolog atoms besides the **url** that identifies *XAK*. It includes two specified properties, thus originating two RDF triples, one for each property. Figure 5.23 shows the

```
?- sparql('http://xmlarmyknife.org/api/rdf/sparql/') />
   hasBody(A) :> hasColor(B) :> item(IND).
```

Figure 5.22: Back-end GNU Prolog/CX query to *XAK*

correspondent SPARQL generated code.

```
SELECT ?id ?hasColor ?hasBody
WHERE {
?id :hasColor ?hasColor.
?id :hasBody ?hasBody.
}
```

Figure 5.23: Generated SPARQL for the query in Figure 5.22

After the SPARQL generation, the code is sent to *XAK*. In order to successfully communicate with it, the back-end must first encode the query as specified in the SPARQL Protocol for RDF [Cla06] and establish the values of some parameters (Figure 5.24 shows the generated string that is sent over to *XAK*):

- The base *url* is directly obtained from the query;
- The request method is `GET`, so the query is encoded and sent as an `HTTP GET` request;
- The request parameters used are *query* and *default-graph-uri*. The first identifies the query and the second the location of the target ontology. At this point, both must be defined in the back-end Prolog knowledge base as facts;
- Prefix value. Should also be defined as a fact;
- The response format is omitted so that the default SPARQL Query Results XML Format is used;

If a successful query response code is returned, a file with the solutions is received. This file is in the SPARQL Query Results XML Format [BB06] (See section 5.1.1) and includes one solution. This XML file is then parsed

```
GET http://xmlarmyknife.org/api/rdf/sparql/query?default-graph-uri
=http://www.w3.org/2001/sw/WebOnt/guide-src/wine.owl&query=
PREFIX+:+<http://www.w3.org/2001/sw/WebOnt/guide-src/wine%23>
+select+?id+?hasColor+?hasBody+where+{?id+:hasColor+?hasColor+.+
?id+:hasBody+?hasBody}
```

Figure 5.24: Back-end encoded query example

and the solution values are returned as bindings for Prolog variables as illustrated by the last lines in Figure 5.25.

```
?- sparql('http://xmlarmyknife.org/api/rdf/sparql/') />
   hasBody(A) :> hasColor(B) :> item(IND).
   .
A ='http://www.w3.org/2001/sw/WebOnt/guide-src/wine#Medium'
B ='http://www.w3.org/2001/sw/WebOnt/guide-src/wine#White'
IND ='http://www.w3.org/2001/sw/WebOnt/guide-src/wine#SelaksWine'?;
(4 ms) no
```

Figure 5.25: GNU Prolog/CX query to *XAK* and the returned solution

The solution presents only one individual, **SelaksIceWine**, and the values **Medium** and **White** for properties *hasBody* and *hasColor* respectively. This means the whole ontology only has one individual that has those two properties defined. Figure 5.26 shows the definition of **SelaksIceWine** found in the ontology.

So there exists only one individual that has values for both *hasColor* and *hasBody* properties. Let us try a more general query and ask for all the individuals that have the *hasFlavor* property and what its value is. As there are several individuals with this property defined (44), the Prolog *backtracking* mechanism is used to fetch all the solutions, one at a time. Figure 5.27 shows the query and the first three solutions.

Naturally, the property *hasFlavor* is defined in most of the individuals of the *Wine* ontology. As can be seen in Figure 5.27, the first returned solutions are the wines *Corbans PrivateBin Sauvignon Blanc*, *Chateau De Meursault Meursault* and *PeterMccoy ChardonnayQ*. The first has a *Strong* flavour and

```
<IceWine rdf:ID="SelaksIceWine">
  <locatedIn rdf:resource="#NewZealandRegion" />
  <hasMaker  rdf:resource="#Selaks" />
  <hasFlavor rdf:resource="#Moderate" />
  <hasBody   rdf:resource="#Medium" />
  <hasColor rdf:resource="#White" />
</IceWine>
```

Figure 5.26: OWL Definition of the *SelaksIceWine* wine

```
?- sparql('http://xmlarmyknife.org/api/rdf/sparql/') />
   hasFlavor(F) :> item(I).

F ='http://www.w3.org/2001/sw/WebOnt/guide-src/
    wine#CorbansPrivateBinSauvignonBlanc'
I ='http://www.w3.org/2001/sw/WebOnt/guide-src/wine#Strong' ? ;

F ='http://www.w3.org/2001/sw/WebOnt/guide-src/
    wine#ChateauDeMeursaultMeursault'
I ='http://www.w3.org/2001/sw/WebOnt/guide-src/wine#Moderate' ? ;

F ='http://www.w3.org/2001/sw/WebOnt/guide-src/
    wine#PeterMccoyChardonnayQ'
I ='http://www.w3.org/2001/sw/WebOnt/guide-src/wine#Moderate' ?
```

Figure 5.27: *XAK* example 2, with more than one solution returned

the other two are *Moderate*.

5.4 Results and Conclusions

The XPTO back-end presented in this chapter is still work in progress. With the current capabilities, one can use the expressiveness of Logic Programming to perform basic queries to an ontology via a third party SPARQL Web Service. These capabilities can then be combined with other GNU Prolog/CX data access forms for reasoning over different data repositories. For example, an application can use indifferently local data provided by the XPTO engine, external data through the SPARQL back-end and data resid-

ing in a relational data base accessed using ISCO [AN06].

Although no proper benchmarks were measured, the experimental work revealed no particular performance issues on the back-end side, which means that practically only the *XAK* connection will introduce some latencies. For instance, the complete processing time for the *Wine* examples are no longer that 20 milliseconds. The first (1 solution) takes 4 ms, and the second (44 solutions) takes 12 ms to return all of them. Note, however, that the generation of SPARQL is currently done in a *per-query* basis. One important feature to be implemented in future work is to allow the generation of SPARQL code for a composite (e.g. conjunction) of GNU Prolog/CX queries.

The communication process can also be improved. For instance, the implementation of a query cache, eventually backed up by an relational database, in order to avoid to establish, in the same program, redundant connections that asks for previously sent queries.

Chapter 6

Conclusion

When the research for this work began, the Semantic Web was emerging as a hot topic in many conferences and workshops around the globe. Nowadays, only two years later, we already have a few Semantic Web yearly conferences and almost every other conference about Internet technologies contains a Semantic Web workshop. The same happened with documentation, where the number of books, articles and Web sites about the Semantic Web are much larger today then they were two years ago. This means that the Semantic Web concept is growing at a great speed and, hopefully, will help to improve the Internet as it is today.

Being on board the "Semantic Web train" gave us great opportunities: we followed the development of technologies like OWL and SPARQL from a very early stage and we had the opportunity to communicate and share thoughts with people actively working on the W3C document specifications about Semantic Web technologies.

The motivation for this work envisioned the development of an information system for the Semantic Web with capabilities to, with some work, evolve into an information hub capable of merging any kind of data from any kind of source in a way it could be used to retrieve and reason over knowledge.

One of our goals was the building of a contextual logic programming framework for the Semantic Web, in which reasoning for documents described by ontologies could be carried out. Within a reasonable set of limitations, this was achieved with the implementation of XPTO. It can parse and represent OWL ontologies from a perspective of contextual logic programming, where access to this system provides a way for reasoning over a previously loaded ontology by means of logic programming.

The loaded ontology is represented by several GNU Prolog/CX units that map all the information, entities and relations present in the ontology to a

GNU Prolog/CX point of view. Querying can then be performed through the use of contexts that include the units in the representation using the operator '/>'.

The fact that it currently does not support a well-defined OWL sub-language in its fullness is one of its main limitations. Another major shortcoming of the current state of the work is that it cannot presently work with more than one ontology at a time. The objective is to be able to load an arbitrary number of ontologies and perform mixed queries over these. The parser works, but it is not very robust yet nor can it handle large ontologies.

The other major goal was the development of a back-end for the SPARQL language. This has been accomplished and a sub-system capable of mapping GNU Prolog/CX to SPARQL queries was implemented. With the current capabilities, one can use the expressiveness and abstraction power of logic programming to perform basic queries to an ontology via a third party SPARQL Web Service.

One of the major difficulties was the choice of the Semantic Web technologies to be used and supported by the system. As of this writing there are no standard OWL query languages yet, the closest thing being SPARQL, which is itself being worked on with the intent of designing such a standard. More importantly, it is an RDF query language, which creates some limitations when used to query ontologies described in OWL. Despite all the efforts being made, it is a matter that is still work in progress with many unsettled issues [Bij06].

The base of the Semantic Web architecture is reasonably established as a starting point, i.e, how the Semantic Web information will be represented with W3C standard recommendations such as OWL and RDF. However, the fact that the top layers of the Semantic Web architecture are still suffering changes and the fact that SPARQL just dropped a step back from W3C candidate recommendation to working draft clearly reflects the amount of work that has yet to be done yet for establishing standards in querying the Semantic Web.

6.1 Future Work

Along the road we had to make tough decisions and move along in order to achieve the goals we set out for. This process necessarily originated some limitations in our work that we marked as subjects to be improved upon, as future work:

- Supporting a well-defined OWL sub-language is necessary in order to

provide reliable, trusted semantic Web agents which will be usable in wider application scenarios. We are working towards providing correct OWL DL compatibility at the reasoning level over the internal representation;

- A crucial goal is to provide the core with capabilities for working with several ontologies at a time. Although it is not relevant for the purpose of this work, it is an essential feature for any Semantic Web application software and we intend to use GNU Prolog/CX's versatile modularity mechanisms to effectively deal with this issue. In order to achieve this some changes will have to be made to the representation. These include changing the name of the units (and unit files) to avoid name clashes, and altering the query method to take into account the several ontologies. The unit `individuals` will also have to be changed to guarantee the separation of individuals of the loaded ontologies, possibly becoming several units, one for each loaded ontology;

- The XPTO parser is basic and not very flexible. It should be more efficient and more robust in order to handle larger ontologies;

- The back-end generation of SPARQL is currently done in a *per-query* basis. One important feature to be implemented in future work is to extend the mapping engine to allow the generation of SPARQL code for a composite (e.g. conjunction) of GNU Prolog/CX queries;

- The back-end communication process can also be improved. For instance, by implementing a query cache system backed up by an internal database. This would allow a more restricted control of the connections in order to not establish, in the same program, redundant connections that ask for previously sent queries;

6.2 Final Considerations

XPTO provides an abstraction representation layer for Web ontologies that can be accessed by logic programs. With the additional back-end capabilities, one can take advantage of the abstraction capabilities of Logic Programming to query ontologies via a third party SPARQL Web Service.

A XPTO represented ontology can then be used with other GNU Prolog/CX data access forms for reasoning over different data repositories. For example, an application can indifferently use local data provided by the XPTO engine, external data through the SPARQL back-end and data residing in a relational data base accessed using ISCO [AN06]. This scenario

brings us some steps closer to our initial intentions about creating an information repository that can access and reason, from a GNU Prolog/CX point of view, over data from different sources.

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.

[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.

[Ant90] Antony Galton. *Logic for Information Technology.* Wiley, 1990.

[Baa99] F. Baader. Logic-based knowledge representation. In M. J. Wooldridge and M. Veloso, editors, *Artificial Intelligence Today, Recent Trends and Developments*, pages 13–41. Springer Verlag, 1999.

[Baa02] Baader and Nutt. *Description Logic Handbook.* Cambridge University Press, 2002.

[Bat05] John Battelle. *The Search.* Nicholas Brealey Publishing, 2005.

[BB06] Dave Beckett and Jeen Broekstra. SPARQL Query Results XML Format. Candidate recommendation, World Wide Web Consortium, 25 December 2006. http://www.w3.org/TR/rdf-sparql-XMLres/.

[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.

[Bij06] Bijan Parsia. Querying the Web with SPARQL. In Enrico Franconi Pedro Barahona, François Bry, editor, *Reasoning Web*, volume 4126 of *LNCS*. Springer, 2006.

[BL01] Tim Berners-Lee. The semantic web. http://www.sciam.com/print_version.cfm?articleID=00048144-10D2-1C70-84A9809EC588EF21, 17 May 2001.

[BN03] Franz Baader and Werner Nutt. Basic description logics. In Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors, *Description Logic Handbook*, pages 43–95. Cambridge University Press, 2003.

[BvHH+05] Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. Owl web ontology language reference. Recommendation, World Wide Web Consortium, 19 October 2005. http://www.w3.org/TR/2004/REC-owl-ref-20040210/.

[Cla06] Kendall Grant Clark. SPARQL Protocol For RDF. Candidate recommendation, World Wide Web Consortium, 6 October 2006. http://www.w3.org/TR/rdf-sparql-protocol/.

[Cla07] Kendall Grant Clark. Rdf Data Access Use Cases and Requirements. Working draft, World Wide Web Consortium, 2007. http://www.w3.org/TR/rdf-dawg-uc/.

[Con07] World Wide Web Consortium. Semantic web, 13 February 2007. http://www.w3.org/2001/sw/.

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

[CR04] Carroll and De Roo. OWL web ontology language test cases. Recommendation, World Wide Web Consortium, 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. http://www.daml.org/. DAML+OIL, 3 February 2007.

[Dav07] Dave Beckett. *Turtle - Terse RDF Triple Language*, 5 March 2007.

[DC00] Daniel Diaz and Philippe Codognet. The gnu prolog system and its implementation. In *SAC (2)*, pages 728–732, 2000.

[Dod06] Leigh Dodds. XML Army Knife. http://xmlarmyknife.org/api/rdf/sparql/query, 5 December 2006.

[FLA07] Cláudio Fernandes, Nuno Lopes, and Salvador Abreu. On querying ontologies with contextual logic programming. In Christine Golbreich, Aditya Kalyanpur, and Bijan Parsia, editors, *OWL: Experiences and Directions 2007*, volume 258 of *CEUR Workshop Proceedings ISSN 1613-0073*, June 2007.

[FN07] Marc Hadley Frystyk Nielsen. Soap: Messaging framework. Recommendation, World Wide Web Consortium, 2007. http://www.w3.org/TR/soap12-part1/.

[Fra02] Enrico Franconi. Description logics tutorial course. Tutorial course, Faculty of Computer Science, Free University of Bozen-Bolzano, Italy, 2002. http://www.inf.unibz.it/ franconi/dl/-course/.

[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.

[Gri04] Grigoris Antoniou and Frank Van Harmelen. *A Semantic Web Primer*. The MIT Press, 11 June 2004.

[HM04] Volker Haarslev and Ralf Möller, editors. *Proceedings of the 2004 International Workshop on Description Logics (DL2004), Whistler, British Columbia, Canada, June 6-8, 2004*, volume 104 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2004.

[HPPSH05] Ian Horrocks, Bijan Parsia, Peter F. Patel-Schneider, and James A. Hendler. Semantic web architecture: Stack or two towers? In François Fages and Sylvain Soliman, editors, *PPSWR*, volume 3703 of *Lecture Notes in Computer Science*, pages 37–41. Springer, 2005.

[Iva01] Ivan Bratko. *Prolog Programming for Artificial Intelligence.* Addison-Wesley Publishers, 2001.

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

[Joh97] John Kelly. *The Essence of Logic.* Prentice Hall, 1997.

[KC07] Graham Klyne and Jeremy J. Carroll. Resource description framework (rdf): Concepts and abstract syntax. Recommendation, World Wide Web Consortium, 2007. http://www.w3.org/TR/rdf-concepts/.

[KMR04] Holger Knublauch, Mark A. Musen, and Alan L. Rector. Editing description logic ontologies with the protégé owl plugin. In Haarslev and Möller [HM04].

[LD01] M. S. Lacher and S. Decker. Rdf, topic maps, and the semantic web. *Markup Languages: Theory and Practice*, 3(3):313–331, December 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 Isabel F. Cruz, Stefan Decker, Dean Allemang, Chris Preist, Daniel Schwabe, Peter Mika, Michael Uschold, and Lora Aroyo, editors, *International Semantic Web Conference*, volume 4273 of *Lecture Notes in Computer Science*, pages 501–514. Springer, 2006.

[MM04] Frank Manola and Eric Miller. RDF Primer. Recommendation, World Wide Web Consortium, February 2004. http://www.w3.org/TR/2004/REC-rdf-primer-20040210/.

[MvH05] Deborah L. McGuinness and Frank van Harmelen. OWL Web Ontology Language Overview. Recommendation, World Wide Web Consortium, 19 October 2005. http://www.w3.org/TR/owl-features/.

[NB03] Daniele Nardi and Ronald J. Brachman. *An Introduction to Description Logics.* The Description Logic Handbook: Theory, imple. Cambridge University Press, Cambridge UK, 2 edition, 2003.

[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. Candidate recommendation, World Wide Web Consortium, 25 July 2006. http://www.w3.org/TR/2006/CR-rdf-sparql-query-20060406/.

[PSH07] Peter F. Patel-Schneider and Ian Horrocks. Owl 1.1 web ontology language overview. Technical report, 20 February 2007. http://www.webont.org/owl/1.1/.

[RC07] Jean-Jacques Moreau Roberto Chinnici. Web services description language: Core language. Working draft, World Wide Web Consortium, 2007. http://www.w3.org/TR/wsdl20/.

[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.

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

[SP04] Evren Sirin and Bijan Parsia. Pellet: An owl dl reasoner. In Haarslev and Möller [HM04].

[Ste94] Sterling and Shapiro. *The Art of Prolog.* MIT Press, 1994.

[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. Recommendation, World Wide Web Consortium, February 2004. http://www.w3.org/TR/2004/REC-owl-guide-20040210/.

[TF06] François Bry Tim Furche, Benedikt Linse. RDF Querying: Language Constructs and Evaluation Methods Compared. In Enrico Franconi Pedro Barahona, François Bry, editor, *Reasoning Web*, volume 4126 of *LNCS*. Springer, 2006.

[Tho04] Thomas Passin. *Explorer's Guide to the Semantic Web*. Manning Publications, 2004.

[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/TheaOWLLib/, 19 January 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.