# Contextual Logic Programming for Ontology Representation and Querying

Nuno Lopes, Cláudio Fernandes, and Salvador Abreu

Universidade de Évora

**Abstract.** The system presented in this paper aims at using Contextual Logic Programming as a computational hub for knowledge modeled by web ontologies and enable querying that representation. The components required to behave as a SPARQL query engine are explained and examples of semantic integration of different sources are shown.

## 1 Introduction

The Semantic Web [6] topic currently represents one of the most active and exciting research areas in computer science. The standard web page provides data oriented for human comprehension, which means a computer agent cannot easily reason about that information. The Semantic Web is a natural evolution of the Internet and, hopefully, will provide the foundations for intelligent systems and agent layers over the World Wide Web.

One important step towards the fulfilling of this vision is the emergence of systems that cannot only understand and reason over Semantic Web documents but also retrieve and process knowledge of multiple information sources. This represents the motivation and purpose of our work which is to use contextual constraint logic programming [2] as a framework for Semantic Web agents, in which knowledge representation and reasoning for ontology documents can be carried out. As such, we adopted the framework Prolog/CX partly described in [3] which makes use of persistence and program structuring through the use of contexts [2]. Throughout this paper, we describe a prototype implementation of a Semantic Web system with three main components:

- A core that is capable of representing web ontologies,
- A SPARQL agent which can answer SPARQL queries about ontologies,
- A back-end capable of mapping Prolog/CX to SPARQL queries, thereby able to query external Semantic Web agents, returning the results as bindings for logic variables present in a Prolog/CX program.

**Web Ontology Languages:** The Semantic Web is based on existing standard technologies such as XML, RDF and RDF-Schema [14]. Although RDF Schema provides additional modeling primitives, like classes and properties, that enable the hierarchical organization of Web documents, a richer ontology modeling language was necessary. DAML-OIL [8] was then taken as the starting point for the W3C Web Ontology Working Group in defining OWL [15], the language that is aimed to be the standardized and broadly accepted ontology language for the Semantic Web [4]. OWL is defined as an extension of a sub set of the RDF vocabulary and is divided into three species [9]: OWL Lite, OWL DL and OWL Full.

**Query Languages:** An open research issue has been the specification of a standard query language that can access this kind of data. There are a variety wide of Semantic Web query languages [11], ranging from pure *selection languages* with limited expressivity to general purpose languages supporting different data representation formats and complex queries. Among all the possibilities, we chose to follow the W3C working groups proposed standard: SPARQL [17], an RDF query language and protocol.

The work presented herein is an extension of what was described in [7], we explain some of the implementation choices and introduce some real world examples. The remainder of this article is structured as follows: Contextual Logic Programming is briefly approached in Section 2 and, in Section 3 we discuss the knowledge representation and ontology querying using Contextual Logic Programming. Section 4 describes a possible approach to implementing a SPARQL agent using the CxLP framework and the issue of querying remote SPARQL agents from within the CxLP framework is discussed in Section 5. Section 6 presents examples of use for the implemented system. Finally, Section 7 provides initial conclusions and possible directions for future research.

## 2 Contextual Logic Programming

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 [2] provide a revised specification for CxLP, which emphasizes the OOP aspects by means of a stateful model, allowed by the introduction of unit arguments. We now informally focus on some aspects of CxLP, namely parametric units; a more complete specification can be found in [2].

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 is 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 [2, 3].

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:[1]

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

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

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

**Current context:** :< C, unifies C with the current context;

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

## 3 System architecture

The implemented system is divided in three parts: the core, a front-end (FE) SPARQL agent and a back-end (BE) that maps Prolog/CX to SPARQL queries. The core system is responsible for representing the ontology, the FE enables the resolution of queries expressed in SPARQL and the BE allows the core (and the FE) to query other SPARQL web services. This architecture is represented in Figure 1. By integrating the core, FE, BE and other Logic Programming frameworks namely ISCO [3], the system will be able to access several heterogeneous sources of information: the ontology, other SPARQL agents or web services and relational databases.

The main objective of the core system is to represent web ontologies with CxLP tools. After an ontology is transformed into Prolog/CX units, the capabilities of that representation are that of pure Prolog with modular program structuring. For instance, we can build a front end that acts as a SPARQL web agent which can receive a SPARQL query over a known ontology, process it against the internal representation and respond with the solution. This representation can also be used to map Prolog goals to SPARQL queries and collect the results as logic variable bindings. These approaches are further discussed in Sections 4 and 5.

### 3.1 Ontology representation

Ontologies are represented using units: there will be one unit that indicates the elements (classes and properties) of the ontologies, another unit for individuals and one for each OWL class and property. This is illustrated in Figure 2.

The individuals and their property values are represented in the unit `individuals`. This unit stores, for each individual, the class it belongs to and, for each of the individual properties, its value.

Each class and property is defined in a unit named after the class or property. Further information about each of this objects, such as hierarchy and restrictions, can be found in its unit.

[1] For a more detailed and formal description, the reader is referred to [2].

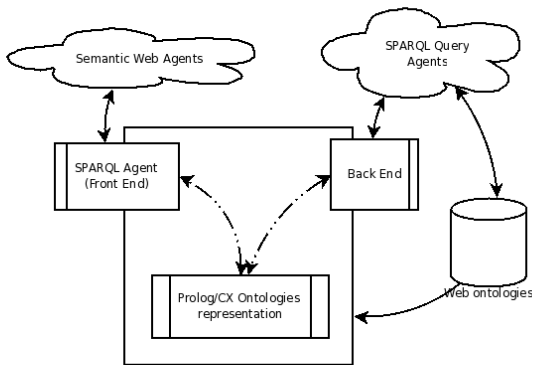

**Fig. 1.** System architecture

The set of known ontologies are represented in a unit named `ontologies` which lists the classes and properties of each loaded ontology. Each property and class listed in this unit can then be accessed in a uniform manner using the operator `/>`. This operator is defined as a context extension operation, i.e., based on the unit name it constructs a new context in which to evaluate the goal.

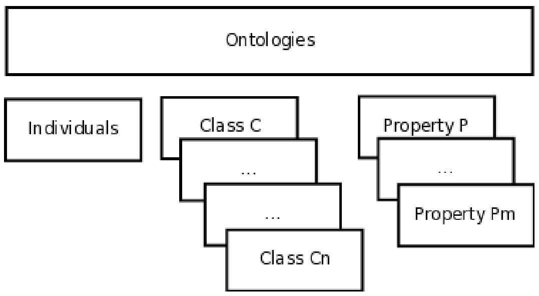

**Fig. 2.** Ontology representation schema

*Ontology Unit* This unit represents the ontology information: namespaces, 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 its value and, for classes and properties, simply the ontology name and the class or property name.

The information in this unit may be used to query which units belong to the ontology, thereby providing access to all the individuals in the ontology.

*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 its value, given the individual name that shall be retrieved previously from the context.

*Class Units* These units will represent the classes of the ontology and all information relevant to that class. The information 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.

*Individuals Unit* This unit is the unit that contains all the individuals, its properties and the information about individual relations. The individuals properties are stored as triples, much in the manner of RDF. These properties are defined in the predicate `property/3`. The first argument of this predicate indicates the name of the individual, the second indicates the property and the third argument contains the value of the property for that individual. All the individuals, along with their class, are listed in the predicate `individual_class/2`. 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.

There may also be present predicates for defining individual relations, such as `differentFrom/2` and `sameAs/2`, each with individual names as their arguments. These indicate, respectively, that the individuals referred are different or the same [15]. 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.

## 3.2 Querying an ontology

The most direct way of retrieving the class individuals is to use the goal `item/1` as shown in Figure 3. There is also a goal `item/0` that has the exact behaviour of `item/1` but has no direct arguments, this predicate, when used with the predicate units in the query will allow to access the property values ignoring the name of the individual.

The `item/1` goal binds, by backtrack, its argument to each individual of the class. There is also the possibility of querying all the individuals in the ontology by omitting a class in the query.

The value of the properties can be accessed by including the unit that represents the property in the context query. This enables selecting only a subset

```
| ?- 'IceWine' /> item(A).
A = 'SelaksIceWine'
```

**Fig. 3.** Accessing an individual of a class

of the properties. The argument of the property unit will be bound to the value of the property for the corresponding individual, as shown in Figure 4.

```
| ?- 'IceWine' /> hasFlavor(F) :> hasBody(B) :> item(I).
B = 'Medium'
F = 'Moderate'
I = 'SelaksIceWine' ?
```

**Fig. 4.** Accessing individuals and properties

## 3.3 Units for refining ontology queries

We propose a number of units which may be used to form queries. We proceed to briefly describe them.

**individual/1** Including this unit in the context unifies its argument with the individual name in the same manner as `item/1`. Using this unit provides a more explicit query, by indicating we want the individual name and calling the goal `item/0` instead of `item/1`. Use of this unit is shown in Figure 5.

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

**Fig. 5.** individual example

**class/1** If this unit is included in the context it will unify its argument with the class of the matching individual. This is useful to determine the class of the individual when querying the entire ontology, as shown in Figure 6.

**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 7).

```
| ?- /> class(C) :> item(I).

C = 'DessertWine'
I = 'WhitehallLanePrimavera' ?
```

**Fig. 6.** class example

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

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

**Fig. 7.** 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 8.

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

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

**Fig. 8.** 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 [17].

## 3.4 Native Prolog query representation

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

– 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.* This allows the user to choose which properties will be present in the generated predicate.

The generated Prolog representation is listed in Figure 9.

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

**Fig. 9.** 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 and thus the user is unable to retrieve its value with these predicates, using this method.

## 4 A SPARQL agent in CxLP

SPARQL is a Candidate Recommendation for a RDF query language [17]. It is under continued development towards becoming the standard query language for the semantic web [11] and although it is mainly used to query RDF graphs, it can also be used to query an RDF Schema or OWL ontology on the individual and properties level [13, 16, 17].

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

The developed system is using SPARQL to query an ontology, allowing access to properties and resulting in individuals and property values.

The implemented SPARQL parser follows the specifications of the language defined in [17] and the results are returned in XML, the format of which is specified in [5]. The parser constructs a Prolog/CX context representing the query; this context is then activated by sending a message to calculate the output and display the resulting XML form. This specification allows our system to be easily made available trough a web service.

SPARQL has 4 types of queries: `select`, `ask`, `construct` and `describe`. The `select` query is used to retrieve the values of the properties and individuals. `Ask` simply returns a boolean answer depending on the veracity of the query. The `construct` and `describe` are not currently implemented as they would return data as RDF graphs.

The following sections briefly describe the SPARQL query language, the resolution of queries and the XML output of the system.

## 4.1 SPARQL and mapping examples

The mapping process (SPARQL parser) transforms a SPARQL query into a Prolog/CX context. The execution of this context will bind the variables present in the query with the results.

The context has a similar structure to the SPARQL query, consisting of the following parts, each of which being a parametrized unit:

**prefix** indicates the default prefix;
**from** specifies the RDF dataset to query;
**select** lists the variables that should be present in the output;
**where** restriction conditions;
***Modifiers*** if present, these modifiers will change the number of results (`limit` and `offset`) and/or their order (`order by`).

The parser receives as input a SPARQL query, shown in Figure 10, and returns the context to be executed (Figure 11).

```
SELECT
       ?flavor ?body
WHERE {
   ?t :hasFlavor    ?flavor  .
   ?t :hasBody      ?body  .
}
```

**Fig. 10.** Query example (simple select)

## 4.2 Query resolution system

The query resolution is triggered by evaluating the goal `item` in the context returned by the mapping process. This is akin to sending the message `item` to an object. The core unit in this process is the unit `triple/1` which is responsible for instantiating the variables in the query by accessing the internal representation of the ontology.

The *Modifiers* will alter the query results, their order or number. If no modifiers are present in the query the unit `all` will be included in the context meaning that all the possible bindings will be returned.

```
1 [ all,
2   where([set([
3               triple(A,hasFlavor,B),
4               triple(A,hasBody,C) ])
5        ]),
6  select([flavor=B,body=C]),
7  vars([flavor=B,body=C,t=A]),
8  defs ]
```

**Fig. 11.** Results of the query example

Also, currently, the `from` clause has no effect since the instantiation is done with an already loaded ontology.

**triple/3** The core unit in this process is the unit `triple/3` which is responsible for instantiating the variables in the query by accessing the internal representation of the ontology. The implementation of this unit is shown in Figure 12. It generates one query to the system core for each property that appears in the SPARQL query. The pattern in line 3 of Figure 11 will generate the following query:

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

The argument of the `item/1` goal will be instantiated with the name of the individual (in this case the variable `I`). The arguments of the unit `property` are the name of the property being queried and the value of that property for the returned individual. Using the `property` unit to query the internal representation has the advantage of being able to perform the query using a variable in the position of the property name (instead of "hasFlavor" in the example).

The results of this query will be bound to the representation of the variables present in the SPARQL query.

```
:- unit(triple(S, P, O)).

item :-
        /> property(P,O) :> item(S).
```

**Fig. 12.** Unit triple

Also, currently, the `from` clause has no effect since the instantiation is done with an already loaded ontology.

"A pattern solution can then be defined as follows: to match a basic graph pattern under simple entailment, it is possible to proceed by finding a mapping from blank nodes and variables in the basic graph pattern to terms in the graph being matched; a pattern solution is then a mapping restricted to just the variables, possibly with blank nodes renamed. Moreover, a uniqueness property guarantees the interoperability between SPARQL systems: given a graph and a basic graph pattern, the set of all the pattern solutions is unique up to blank node renaming." in [17].

Each basic operation is represented as a unit.

All the units that are present in the context generated by the SPARQL parser will answer to the goal `item/0` or `item/1` (in case of the unit returning a bound solution). Each unit will then perform the operation it represents based on its arguments and on the result of the parent context.

The units that alter the query results (the *Solution Modifiers*), such as `order by`, `limit` and `distinct` fetch all the bound variables from the parent context collecting them in a list. They them perform its operation on over the elements of the list thus achieving a new list with the results. This will be the final list to be presented as XML.

## 5 Mapping Prolog to SPARQL Queries

The purpose of having web ontologies represented in a logic contextual form is to create a mechanism to access and work over those ontologies. So far, an ontology mapping engine for local reasoning was introduced. However, for increased flexibility and functionality, one could consider of merging the reasoning of the system internal knowledge base with external ontologies provided by external Semantic Web services. To achieve this goal, a back end was developed that is capable of communicating with SPARQL web agents. This enables writing Prolog/CX programs to reason simultaneously over local and external ontologies.

### 5.1 Architecture

The back end engine provides additional means to query external Semantic Web services in SPARQL. Although it can be viewed as a single independent component, the objective is to integrate it with the system in a manner that allows a programmer to reason over external and internal ontologies using the same query syntax and declarative context mechanics as the internal system reasoning. This will allow the system using programmer to query transparently internal and external ontologies and merge their results in the same program.

To achieve this level of functionality, a Prolog/CX to SPARQL engine was developed that has the following level of execution capability:

- Translates a Prolog/CX goal into SPARQL;
- Sends the SPARQL query to the indicated Semantic Web SPARQL service;
- Fetch the XML result file, parse it and return the solutions as Prolog variable bindings using the Prolog/CX backtrack mechanism.

It is necessary to provide additional information in order to query the external agent if the SPARQL protocol [18] 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 may have capabilities for querying ontologies from any given Internet location, such as the `XML Armyknife` Semantic Web service [10] that is used throughout this section to illustrate the back end functionality. The response format can vary from different types like simple HTML for Internet browsing purposes, or the SPARQL Query Results XML Format [5] for agents like ours.

### 5.2 Querying an external SPARQL agent

Accessing different Semantic Web external agents rises the problem of implementing a unique interface that can communicate with all of them. The core of this problem has been addressed by the W3C group, which is working on a SPARQL protocol for web agents communication [18]. This W3C *Candidate Recommendation* describes means of conveying SPARQL queries from query clients to a SPARQL query processing service and returning the query results to the requesting entity. Therefore, and after finding some SPARQL agents that rely on this protocol [10, 1], we decided that at this point, for demonstration and proof of concept purposes, the back-end would also rely on the interface communication described on this document.

**Communication and query solutions** The execution of a back-end query can be described as a three step process. The first is the process of mapping a Prolog/CX query to a SPARQL. The query illustrated in Figure 10 originates the following SPARQL query (Figure 13):

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

**Fig. 13.** generated SPARQL code

After the Prolog/CX translation to SPARQL constructs a query, 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

After correctly encoding the SPARQL query, the back end will start the communication process with the external agent. This represents the *Establish connection* item in the above list and includes the validation of the the Web service:

- Open the communications via `C` sockets;
- Verify if the external host is up and ready for communication.

After the connection is established, the query is then sent through the socket. If everything went well, the external agent response is then received and flushed into a XML file. Then the connection is closed. This represents the remain items in the back end query execution list.

After receiving the response and closing the connection, the response is saved locally in a XML file and the process returns to the Prolog side, where the file is parsed and processed. The XML format that represents the solutions to the query follows the specification described in the SPARQL Query Results XML Format [5]. This file, which contains all the existing solutions for the query, is then parsed and converted into a Prolog List. Finally, the back end will provide each logic solution to the query present in the response file, one at a time if more than one are available, via the backtracking mechanism.

## 6 Examples

### 6.1 Using SPARQL to query a relational database

By using ISCO framework we enable to use SPARQL to query any database. The database used as test in the query example shown in Figure 15 is the database of Universidade de Evora's Information System (SIIUE) [12]. The database contains data relative to the implementation of Academic services. The relation used to query in Figure 15 is represented in Figure 14. It is necessary to define that each field name is prefixed by the name of the relation and an _ forming: `RelationName_FieldName` in order to be able to represent fields with the same name from different relations. This way, the `:student_number` query in line 2 of Figure 15 represents the field `number` of table `student`. The name of the individual, that will be mapped to each tuple in the relation, is the Postgresql internal `OID`[2] of the tuple.

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

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

```
mutable class student.
  id: individual.id. unique.
  number: int. unique.
  institution: institution.id.
```

**Fig. 14.** ISCO definition of the relation aluno

```
select * where {
  ?a :student_number ?c .
  ?a :student_institution ?b .
  FILTER ( ?c > 300 && ?c < 500  )
}
```

**Fig. 15.** using SPARQL to query a relational database

- allow to query more than one relation field. As the query is being translated to the ISCO language it is performing one query to the database for each field present in the SPARQL query. This could be improved by detecting patterns in the query and rewriting it to minimize the number of queries to the database;
- enable filtering the elements before they are retrieved from the relation. In its current stage all the elements are gathered from the relation being queried and filtered later in the `filter` statement.

### 6.2 SPARQL Web service

As another example of the developed system an example web interface was built in order to allow answering of SPARQL queries over the web.

This has a simple user interface in which users can specify the queries and retrieve the results.

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

This form of query method has some shortcomings:

- the core system has to load the ontology for each query it is made (which takes some time);
- the ontology over which the query will be performed is already integrated with the system but, in this case, there is only the possibility of querying the loaded ontology.

The implemented example works under the second assumption. The ontology is integrated with the system and the SPARQL query will be performed over that ontology.

## 7 Conclusion

The system we described and implemented provides a representation abstraction layer for web ontologies that can be accessed by logic programs.

The selected web languages, SPARQL and OWL, have been shown to be appropriate for the scope of our work: to build a working proof-of-concept system which allows us to experiment with Contextual Logic Programming to represent and query ontologies in a way which draws on Prolog's expressiveness as well as the powerful composition mechanisms of CxLP.

We illustrated how this representation can be used to develop Semantic Web agents by describing two components: a front-end and a CxLP back-end. There are aspects of other, existing systems that will benefit from the ability to query SPARQL sources: for instance, we are working on integrating the SPARQL back-end into the ISCO [3] system.

**Future Work** Supporting a well-defined OWL sublanguage is necessary in order to provide reliable, trusted semantic web agents which will be usable in wider application sceneries. We are working towards providing provably correct OWL DL compatibility at the reasoning level, over the internal representation. This issue is orthogonal to the rest of the work described herein but it is essential if we expect the system to gain acceptance in the design of SW agents.

The implemented SPARQL agent currently does not cover the full language specification. Although full SPARQL language support is not our immediate intended purpose, we are working towards providing complete support for it.

Another important goal is to provide the core with capabilities to work 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 purport to use CxLP's versatile modularity mechanisms to effectively deal with this issue. This aspect will be the goal of upcoming work.

## References

1. SPARQler. http://sparql.org/sparql.html, 10 October 2006.
2. 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.
3. 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.
4. Grigoris Antoniou and Frank van Harmelen. *A Semantic Web Primer*. The MIT Press, 2004.
5. D. Beckett and J. Broekstra. SPARQL Query Results XML Format. W3C recommendation, W3C, April 2006. http://www.w3.org/TR/rdf-sparql-XMLres/.

6. T. Berners-Lee, J. Hendler, and O. Lassila. The semantic Web. *Scientific American*, 284(5), 2001.
7. Nuno Lopes Cláudio Fernandes and Salvador Abreu. On querying ontologies with contextual logic programming. *OWL: Experiences and Directions Third International Workshop*, June 2007.
8. DARPA. http://www.daml.org/. DAML+OIL, 3 February 2007.
9. M. Dean, G. Schreiber, S. Bechhofer, Frank van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. OWL web ontology language reference. W3C recommendation, W3C, Feb 2004. http://www.w3.org/TR/owl-ref/.
10. Leigh Dodds. XML Army Knife. http://xmlarmyknife.org/api/rdf/sparql/query, 5 December 2006.
11. Tim Furche, Benedikt Linse, François Bry, Dimitris Plexousakis, and Georg Gottlob. Rdf querying: Language constructs and evaluation methods compared. In Pedro Barahona, François Bry, Enrico Franconi, Nicola Henze, and Ulrike Sattler, editors, *Reasoning Web*, volume 4126 of *Lecture Notes in Computer Science*, pages 1–52. Springer, 2006.
12. Joaquim Godinho, Luis Quintano, and Salvador Abreu. Universidade de Évora's Integrated Information System: An Application. In Hans Dijkman, Petra Smulders, Bas Cordewener, and Kurt de Belder, editors, *The 9th International Conference of European University Information Systems*, pages 469–473. Universiteit van Amsterdam, July 2003. ISBN 90-9017079-0.
13. Jena. A Semantic Web Framework for Java. http://jena.sourceforge.net/, 30 November 2006.
14. Frank Manola and Eric Miller. Rdf primer. W3C Recommendation, World Wide Web Consortium, February 2004.
15. Deborah L. McGuinness and Frank van Harmelen. OWL Web Ontology Language Overview. W3C Recommendation, World Wide Web Consortium, February 2004.
16. Protégé. Free, open source ontology editor and knowledge-based framework. http://protege.stanford.edu/, 30 November 2006.
17. Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. Technical report, W3C, 2006.
18. W3C. SPARQL Protocol For RDF. http://www.w3.org/TR/rdf-sparql-protocol/, 6 October 2006.