Representing and Querying Multiple Ontologies with Contextual Logic Programming

Nuno Lopes[1], Cláudio Fernandes[2], and Salvador Abreu[2]

[1] Digital Enterprise Research Institute, National University of Ireland, Galway
nuno.lopes@deri.org
[2] Universidade de Évora
{cff,spa}@di.uevora.pt

Abstract. The system presented in this paper uses Contextual Logic Programming as a computational hub for representing and reasoning over knowledge modeled by web ontologies, integrating the approach with similar mechanisms which we already developed. As a result of its Logic Programming heritage, the system may also recursively interrogate other ontologies or data repositories, providing a semantic integration of multiple sources. The components required to behave as a SPARQL query engine are explained and examples of integration of different sources are shown – in particular, the case of multiple OWL ontologies is discussed.

1 Introduction

The Semantic Web [5] is currently one of the most active and exciting research areas in computer science. Standard web pages provide data adequate for human comprehension, mixing presentation with content, which means that an automated agent cannot easily extract or reason about that information. The Semantic Web is a natural evolution of the World Wide Web which, hopefully, will provide the foundations for intelligent systems and agent layers over the Web.

One important step towards the fulfilling of this vision is the emergence of systems that can, not only understand and reason over Semantic Web documents, but also retrieve and process knowledge originating from multiple heterogeneous information sources. One motivation and purpose for our work is to use contextual constraint logic programming [1] 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 GNU Prolog/CX programming system, partly described in [2] which incorporates persistence and program structuring through the use of contexts [1]. 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, integrating that representation with normal GNU Prolog/CX predicates and modules, in order to make them interoperable.
- A SPARQL agent which can answer SPARQL queries about ontologies,

– A back-end capable of mapping GNU Prolog/CX 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. This back-end is meant to be integrated with the ISCO [2] framework and is similar to the built-in SQL generation.

Web Ontology Languages: The Semantic Web is based on existing standard technologies such as XML, RDF and RDF-Schema [16]. 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 [7] was then taken as the starting point for the W3C Web Ontology Working Group in defining OWL [17], the language that is aimed to be the standardized and broadly accepted ontology language for the Semantic Web [3]. OWL is defined as an extension of a sub set of the RDF vocabulary and is divided into three species [8]: 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 [19], an RDF query language and protocol.

The work presented herein is expanded from what was described in [15, 10, 14], 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 recalled in section 2, in section 3 we discuss the knowledge representation and ontology querying using Contextual Logic Programming. The issue of querying remote SPARQL agents from within the CxLP framework is discussed in section 5. Finally, section 6 provides preliminary conclusions and points out possible directions for future research.

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

– **Thea** [22] - An OWL tool capable of parsing an OWL ontology and representing it using Logic Programming. 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 [23];
- **Racer** [21] - An OWL reasoner and inference engine for the Semantic Web. Implemented in Common Lisp, Racer is able to start multiple reasoners on the local machine and distribute its load among the Racer instances. It also uses query caching, each query sent by clients and each answer to that query obtained through reasoning can be cached by the system;
- **Protege** [18] - A Semantic Web platform that provides tools to construct Web ontologies. It has a plug-in interface [13] that allows integration with Web ontology reasoners such as Racer;
- **Jena** [12] - 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 known as *ARQ*, which allows Jena from within its framework to query an external SPARQL agent and process the returning results;
- **Pellet** [20] - A reasoner for the OWL DL sub-language. It contains a query engine which supports answering queries formulated using SPARQL and supports reasoning with multiple ontologies;
- **F-OWL** - F-OWL is implemented using Flora-2 which is a extension of F-logic, a logic based language with some aspects of Object-Oriented Programming. F-OWL is a rule based ontology inference engine for OWL which makes use of mechanisms of the underlying technology (XSB Prolog) such as tabling for result caching.

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 [1] 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 discussion can be found in [1].

A unit is a parametric module, constituting the program's static definition building blocks. 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 result from the specific units which make up the context. A more extensive description of CxLP may be found in [1, 2].

Some immediate parallels can be made between 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 evaluated 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 are used to modulate the context part of a computation. In a nutshell, when executing a goal G in a context $C = u_1.u_2...$ where each u_i is a parameterized unit, a CxLP Engine will traverse C looking for the first (topmost) unit u_i 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 C which starts with unit u_i. Some of the most frequently 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 in the resulting context;

Context switch: C `:<` G, evaluates goal G in context C, bypassing the current context;

Supercontext: `:^` G, evaluates goal G in the context obtained by removing the topmost unit from the current context. This is useful to layer specialized on top of generic behavior;

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

Calling context: `:>` C, unifies C with the *calling context*, i.e. the context which was active when the present goal was initially evaluated.

In short, GNU Prolog/CX can be used to take an OO approach while retaining the benefits of (Constraint) Logic Programming, namely nondeterminism via backtracking or constraint propagation and the logical variable as a means of carrying incomplete information. Informally, GNU Prolog/CX has been used as one of the base components for building web-based information systems, the other component being ISCO, a layer which provides persistence at the Prolog level by means of a relational database interface.

Collectively, GNU Prolog/CX and ISCO become a flexible mediator framework in which different information sources may be integrated. This flexibility extends to the reasoning abilities of the resulting system as it can rely on a structured *context* to provide a variable theory against which to match goals: the context in which a CxLP goal is being proved provides its semantics. The CxLP context is made up of *units* which are the specific components that determine how a goal is interpreted; it is viable to have units which map to querying a relational

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

database, others which compute their results using regular Prolog predicates and yet others which resort to different mechanisms, for instance querying a web service using SPARQL.

3 System architecture

The XPTO system is designed to achieve a exact representation of OWL ontologies using the methods provided by GNU Prolog/CX. The main motivation of the development of XPTO is the integration with ISCO [2], which allows accessing several heterogeneous sources of information: the ontology, other SPARQL agents or web services, the outcome of Prolog computations as well as relational databases.

The initial implementation of the XPTO system is divided in three parts: the core, a SPARQL front-end agent (FE) and a back-end (BE) that maps GNU Prolog/CX to SPARQL queries as represented in Figure 1.

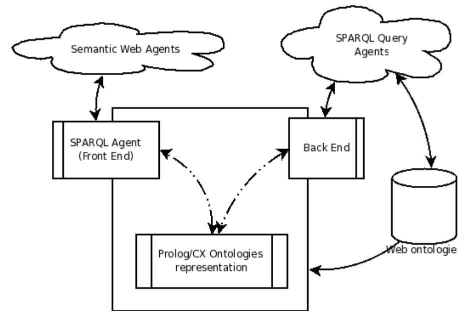

Fig. 1. System architecture

3.1 Representation of ontologies

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 arrangement is illustrated in Figure 2.

The individuals, along with 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.

The set of known ontologies are represented in a unit named `ontology` 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 `.>` (See Section 3.3). This operator is defined as a context extension operation,

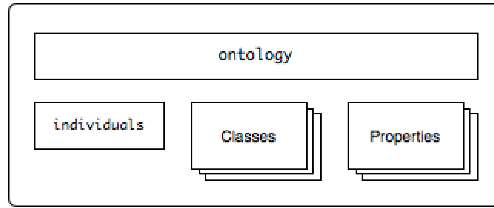

Fig. 2. Ontology representation schema

i.e., based on the unit name it constructs a new context in which to evaluate the goal.

In this paper, the ontology examples are taken from the Wine ontology [24] and the queries are also performed over this ontology. Next are described in more detail, along with partial examples[2], each element of the representation:

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.

It is possible to represent several ontologies, i.e. to load more than one ontology into XPTO, and thus querying multiple ontologies also becomes possible. To achieve this, it was necessary to alter both the naming of units and unit files, that is an evolution from the description provided in [14] (where property and class units where named after the property or class). To avoid name clashes, these units were given an internal unique name generated with a prefix `property_` (or `class_`) and a sequential number. A mapping between the unit name and the ontology element is kept in the `ontologies` unit in the predicate that lists the currently available properties and classes: `prop/3` and `class/3` respectively. A simple example follows:

```
prop(ontology_1, 'http://example.org/'#someProp, prop_1).
class(ontology_1, 'http://example.org/'#someClass, class_1).
```

The internal representation of the ontology elements stores the base namespace along with the name of the element, this choice provides a safe behavior when dealing with elements of different ontologies with the same name. An example of this unit is presented in Figure 3.

[2] Due to space considerations, we omit the prefix http://www.w3.org/TR/2003/PR-owl-guide-20031209/ from the ontology URIs, which get shortened to `&wine;` and `&food;`, respectively

```
:- unit(ontologies).

ns(ontology_1, xmlns, '&wine;').
ns(ontology_1, xmlns:vin, '&wine;').

header(ontology_1, rdfs:comment, ' An example OWL ontology').
header(ontology_1, owl:priorVersion,
       'http://www.w3.org/TR/2003/CR-owl-guide-20030818/wine').

class(ontology_1, '&wine;'#'Wine', class_1).
class(ontology_1, '&wine;'#'Vintage', class_2).

prop(ontology_1, '&wine;'#locatedIn, prop_1).
prop(ontology_1, '&wine;'#adjacentRegion, prop_2).
```

Fig. 3. Unit "ontologies" (partial)

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. This can be seen in Figure 4.

These properties also define the method to access its value, given the individual name that shall be retrieved previously from the context.

```
:- unit(prop_1(A)).

object(rdf:type('http://www.w3.org/2002/07/owl'#'TransitiveProperty')).
domain_name('http://www.w3.org/2002/07/owl'#'Thing').
range('&wine;'#'Region').
type(object).

item :- item(_).
item(B) :- :^ item(B),
           access_property(B, prop_1, A).
```

Fig. 4. Unit "prop_1" (partial)

Class Units These units will represent the classes of the ontology and all information relevant to that class (as shown in Figure 5). 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.

```
:- unit(class_1).

axiom(subClassOf('&wine;'#'Wine','&food;'#'PotableLiquid')).

subClassOf('&food;'#'PotableLiquid').

descriptor(restriction('&wine;'#hasMaker,
                       constraint(allValuesFrom, '&wine;'#'Winery'))).

superClassOf('&wine;'#'LateHarvest').
superClassOf('&wine;'#'EarlyHarvest').
superClassOf('&wine;'#'DessertWine').

unit_name(class_1).

class_name('&wine;'#'Wine').
```

Fig. 5. unit "class_1" (partial)

Individuals Unit This unit is the unit that contains all the individuals, its properties and the information about individual relations. A partial example of this unit is presented in Figure 6. 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 [17]. 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.

```
:- unit(individuals).

individual_class('Year1998', '&wine;'#'VintageYear').
individual_class('USRegion', '&wine;'#'Region').

property('MedocRegion', locatedIn, '&wine;'#'BordeauxRegion').
property('LoireRegion', locatedIn, '&wine;'#'FrenchRegion').

differentFrom('OffDry', '&wine;'#'Dry').
differentFrom('OffDry', '&wine;'#'Sweet').
```

Fig. 6. Unit "individuals" (partial)

3.2 Building the representation

In order to query an ontology, that ontology must first be transformed and loaded into the system, in a way similar to the compilation process. This results in the ontology being represented as the structure described in Section 3.1.

XML parse: In this step the ontology is handled as a plain XML file and therefore parsed using a standard XML parser.

The selected parser was the "The Expat XML Parser".[3] The parser creates a Prolog term that is an accurate representation of the XML file, and apart from the possible comments in the ontology file, there is no loss of information in this transformation.

Name analysis: The next process is to match the term created by Expat and build a dictionary with all the information we need to generate the units and predicates that will represent the ontology. The body of the ontology is mapped focusing mostly on classes, properties, individuals and relations between these elements. Ontology headers are also stored to be included in the ontology definition unit.

[3] http://expat.sourceforge.net/

Unit generation: At this stage, the system has all the information needed to generate the units that will represent the ontology, where each class and property will rise a different unit. Those elements are available in the symbol table, so the mapping engine must walk through it, and for every item generate a unit according with the structure discussed in section 3.1.

Compiling and loading the units: In GNU Prolog/CX, an unit must first be compiled and loaded before one can execute its predicates. This means the system, after parsing an ontology and generating the units, must compile and load the Prolog file that contains each unit. This is achieved using the *dynamic loading* of GNU Prolog/CX. Compiling and loading the units represents the last step of the whole core system process, which starts by parsing the ontology, then the name analysis, unit generation, and finally the compilation and loading of the units.

3.3 Querying an ontology

The new ontology representation schema implies a new query format: it is necessary to explicitly include the namespace of the property or class in the query. This may be a logic variable (as shown in Figure 7) which can be used to either specify or inquire about the namespace part.

```
| ?- N#someClass .>  someProp(F) :> item(I).

F = 'http://example.org/'#'somePropValue'
I = 'http://example.org/'#'someIndividual'
N = 'http://example.org/' ?
```

Fig. 7. Query example

The most direct way of retrieving the class individuals is to use the goal `item/1` as shown in Figure 8. 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 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 9.

```
| ?- N#'IceWine' .> item(N#A).
A = 'SelaksIceWine'
N = 'http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine' ?
```

Fig. 8. Accessing an individual of a class

```
| ?- N#'IceWine' .> hasFlavor(N#F) :> hasBody(N#B) :> item(N#I).
B = 'Medium'
F = 'Moderate'
N = 'http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#'
I = 'SelaksIceWine' ?
```

Fig. 9. Accessing individuals and properties

3.4 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 10.

```
| ?- .> individual(N#I) :> item.
I = 'WhitehallLanePrimavera'
N = 'http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine' ?
```

Fig. 10. 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 11.

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 12).

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 13.

```
| ?- .> class(_N#C) :> item(_N#I).
C = 'IceWine'
I = 'SelaksIceWine' ?
```

Fig. 11. class example

```
| ?- N#'IceWine' .> individual(N#I) :> property(N#P,N#V) :> item.
I = 'SelaksIceWine'
N = 'http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine'
P = locatedIn
V = 'NewZealandRegion' ?
```

Fig. 12. property 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 [19].

3.5 Native Prolog query representation

To make simple queries easier for Prolog programmes, there is the possibility of creating 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

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

Fig. 13. all example

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 14.

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

Fig. 14. 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 [19] that is under continued development towards becoming the standard query language for the Semantic Web [11].

This chapter describes the Front End (FE), the component of the application dedicated to SPARQL query resolution: it allows for the possibility of querying the internal representation of the ontology using the SPARQL query language. The FE is split into 3 parts: the parser, the query resolution and the returning of the results as XML. The implemented SPARQL parser follows the specifications of the language defined in [19] and the results are returned in XML as specified in [4].

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 parser constructs a GNU 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.

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

4.1 Query representation

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

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

Representation of variables and resources SPARQL variables appear in the generated context for the query using the `PrologVariable` representation, enabling a simple access to the value of the variable or direct instantiation of an unbound variable. This representation can be seen in the GNU Prolog/CX context shown in Figure 16.

There are some other structures needed to display the results: it is necessary to store the name of the variable in the SPARQL query in order to return it in the results. To achieve this, all the variables in the SPARQL query are stored in a list that will be the argument of the unit `vars/1`. The elements of this list are in the format `SparqlVariableName = PrologVariable` that represents the assignment of each SPARQL variable present in the query to a logic variable. `PrologVariable` will start unbound and, as the context is resolved, will be instantiated with the solutions it may have.

Resources are represented using Prolog terms (for prefixed IRIs) or atoms in the case of complete IRIs. For example, `<http://example.org/book/book1>` will be is represented as `'http://example.org/book/book1'`.

For prefixed names are represented as Prolog compound terms of arity 2 with the functor ':'. Assuming the following prefix definition: `PREFIX dc: <http://purl.org/dc/elements/1.1/>`, the prefixed name `dc:title` is represented as `dc:title`. If the prefix name is empty the atom `''` will be used to represent it.

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

Context structure The parser receives as input a SPARQL query, shown in Figure 15, and returns the context to be executed (Figure 16). The example query presented in Figure 15 is a `select` query containing two basic graph patterns with a shared variable: `?t` and the context produced by the parser in shown in Figure 16.

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

A context is represented by a Prolog list containing unit names. The first element of the list will be the unit that first tries to evaluate the goal upon execution. The individuals and property values are gathered from the units in a higher position in the context. This way in the final positions of the list are

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

Fig. 15. Query example (simple select)

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

Fig. 16. Generated context (partial) for the query in Figure 15

found the units `select/1` (in the case of a select query) and `vars/1`. These units contain in their arguments a list of variables and will allow any unit in the context to access either all the variables in the context or the selected variables.

The generated parser was tested against a set of the most common SPARQL queries and against SPARQL syntax examples present in [19]. Although there are cases not being handled by the resolution system the parser itself is able to generate the correct context for the query.

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

Although not presented in Figure 15 the units `from`, `prefix` and `base` are always present in the generated context (even if absent in the query). If any of these keywords are not present in the query the corresponding unit will contain an empty list (this can be noted in the complete context presented in Figure 18).

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

The unit that is the core of the query engine is `triple/3`. This unit will represent a simple graph pattern and the arguments of the unit are respectively the `subject`, `property` and `object` of the graph pattern. This unit will be responsible for accessing the dataset in order to bind the variables that may appear in the arguments or test if the pattern has a solution.

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

Fig. 17. Query example

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

Fig. 18. Context generated for the query in Figure 17

4.2 Result extraction and output

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.

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 19. It generates one query to the system core for each property that appears in the SPARQL query. The pattern in line 3 of Figure 16 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. 19. Unit triple

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

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.

XML output The output of the SPARQL query execution is an XML document with a `sparql` element. This element has then two sub-elements: the `head` element and the `results` element (always shown in this order).

The first element for a `select` query is a list of all the variable names present in the SPARQL query. For an `ask` query (that only returns a boolean) no elements are present. The second element (`results`) is a list of `result` elements. The `result` element has two boolean attributes: ordered and distinct, that are always specified. They indicate, respectively, if the list of results is ordered and if the elements are all different. Its value is defined by the presence or absence of the modifiers `distinct` and `order by` in the SPARQL query. The resulting XML of the query in Figure 15 is shown in Figure 20.

```
<sparql>
  <head>
    <variable name="flavor"/>
    <variable name="body"/>
  </head>
  <results ordered="false" distinct="false">
    <result>
      <binding name="flavor">"Medium"</binding>
      <binding name="body">"Moderate"</binding>
    </result>
  </results>
</sparql>
```

Fig. 20. XML output of the query example

5 Mapping Prolog to SPARQL Queries

The back-end component of XPTO behaves as a mapping system from GNU Prolog/CX queries to SPARQL which is capable of communicating with SPARQL endpoints. 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. Figure 21 illustrates the architecture of XPTO with the integration of the SPARQL back-end component.

To query the external SPARQL endpoints the next steps are followed:

1. The back-end translates a GNU Prolog/CX goal into a SPARQL query;
2. The resulting SPARQL query is sent to the appropriate SPARQL web service;
3. The solution set, described by an XML document, is fetched and the corresponding logic variables which occurred in the original query are nondeterministically bound.

It is necessary to provide additional information in order to query the external agent if the SPARQL protocol [6] 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 [9]. The response format can vary from different types like simple HTML for Internet browsing purposes, or the SPARQL Query Results XML Format [4] for agents like ours.

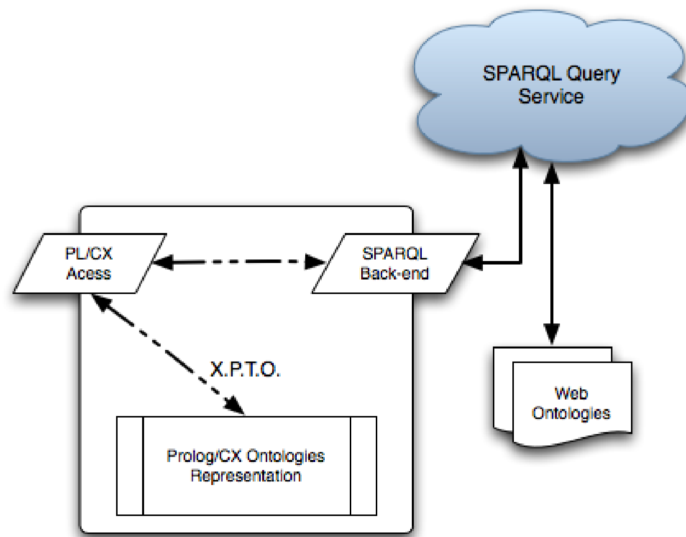

Fig. 21. System architecture

5.1 Generating SPARQL SELECT queries

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

```
QUERY := sparql(URI) /> N1#P1 ... Nn#Pn :> ITEM
         URI  := URL
         P    := property(VALUE) or where(PROP, VALUE)
         ITEM := item(INDIVIDUAL)
```

Fig. 22. Back-End Query Definition

On the left side of the main operator `'/>'` the external SPARQL endpoint URI is specified and, on the right side, the goals and query restrictions. The latter part of the query triggers the query that will be mapped to SPARQL. This translation is obtained by transforming each property present in the query into a triple that relates the property value to the individual.

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. The following query:

```
sparql('uri.org') />  N1#property1(V1) :>
                      N2#where(PROP, 'value2') :> item(IND).
```

will be translated into the triples:

```
(?IND, N1#property1, ?V1)
(?IND, N2#?PROP, 'value2')
```

All unbound Prolog variables represent SPARQL variables in the triples. To state a value in the query and therefore apply a restriction to the solution, a Prolog atomic value can be used to bind a Prolog variable.

If more than one solution is available for the query, all the results are retrieved using the Prolog backtracking mechanism.

To ask for a property name, i.e, to generate a triple in which the property position is a variable, the auxiliary unit `where/2` should be used (as illustrated in the previous 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` clause arguments and the triples will form the sets in the `WHERE` clause.

5.2 Query example

Figure 23 shows an example of a back-end query that asks a SPARQL endpoint to search the *Wine*[4] ontology for all the individuals that have both `hasBody` and `hasColor` properties.

The generated SPARQL query (Figure 24) is then sent to the SPARQL endpoint. In order to successfully communicate with it, the back-end must first encode the query as specified in the SPARQL Protocol for RDF [6] and if a successful query response code is returned, a solution file is received. This file follows the SPARQL Query Results XML Format [4] and includes one solution.

[4] The ontology is accessible in http://www.w3.org/TR/owl-guide/wine.rdf

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

A ='Medium'
B ='White'
N ='http://www.w3.org/2001/sw/WebOnt/guide-src/wine'
I ='http://www.w3.org/2001/sw/WebOnt/guide-src/wine' # 'SelaksIceWine'
```

Fig. 23. GNU Prolog/CX query and solution

It is then parsed and the solution values are returned as bindings for Prolog variables as illustrated by the last lines in Figure 23.

In the present example, the solution presents exactly 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 both of these properties defined.

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

Fig. 24. Generated SPARQL query

6 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 transparently integrating the SPARQL back-end into the ISCO [2] 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 Semantic Web 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.

A performance study, including a proper comparison with related systems, has already been partially performed [14] and will be further expanded as the implementation evolves.

References

1. 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.
2. 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.
3. Grigoris Antoniou and Frank van Harmelen. *A Semantic Web Primer*. The MIT Press, 2004.
4. D. Beckett and J. Broekstra. SPARQL Query Results XML Format. W3C recommendation, W3C, April 2006. Available at: http://www.w3.org/TR/2006/CR-rdf-sparql-XMLres-20060406/.
5. T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284(5), 2001.
6. Kendall Grant Clark. SPARQL Protocol For RDF. Candidate recommendation, World Wide Web Consortium, 6 October 2006. http://www.w3.org/TR/rdf-sparql-protocol/.
7. DARPA. DAML. http://www.daml.org/, 3 February 2007.
8. 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/.
9. Leigh Dodds. XML Army Knife. http://xmlarmyknife.org/api/rdf/sparql/query, 5 December 2006.
10. 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.
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. Jena. A Semantic Web Framework for Java. http://jena.sourceforge.net/, 30 November 2006.
13. Holger Knublauch, Mark A. Musen, and Alan L. Rector. Editing description logic ontologies with the protégé owl plugin. In Volker Haarslev and Ralf Möller, editors, *Description Logics*, volume 104 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2004.
14. 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.
15. Nuno Lopes, Cláudio Fernandes, and Salvador Abreu. Representing and querying multiple ontologies with contextual logic programming. July 2008.
16. Frank Manola and Eric Miller. RDF Primer. W3C Recommendation, World Wide Web Consortium, February 2004.
17. Deborah L. McGuinness and Frank van Harmelen. OWL Web Ontology Language Overview. W3C Recommendation, World Wide Web Consortium, February 2004.
18. Protégé. Free, open source ontology editor and knowledge-based framework. http://protege.stanford.edu/, 30 November 2006.
19. Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. Technical report, W3C, 2006. Available at: http://www.w3.org/TR/2006/CR-rdf-sparql-query-20060406/.
20. Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical owl-dl reasoner. *J. Web Sem.*, 5(2):51–53, 2007.
21. Software Systems Institute. Racer Manager. http://racerproject.sourceforge.net/, 19 February 2007.
22. Vangelis Vassiliadis. Thea OWL Parser for Prolog. http://www.semanticweb.gr/TheaOWLParser/, 12 October 2006.
23. Vangelis Vassiliadis. *Thea A Web Ontology Language - OWL Parser for [SWI] Prolog.* http://www.semanticweb.gr/downloads/Thea%20OWL%20Parser.doc, 19 January 2007.
24. W3C. Wine Ontology. http://www.w3.org/TR/owl-guide/wine.rdf, 22 July 2006.

References

1. 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.
2. 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.
3. Grigoris Antoniou and Frank van Harmelen. *A Semantic Web Primer.* The MIT Press, 2004.

4. D. Beckett and J. Broekstra. SPARQL Query Results XML Format. W3C recommendation, W3C, April 2006. Available at: http://www.w3.org/TR/2006/CR-rdf-sparql-XMLres-20060406/.
5. T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284(5), 2001.
6. Kendall Grant Clark. SPARQL Protocol For RDF. Candidate recommendation, World Wide Web Consortium, 6 October 2006. http://www.w3.org/TR/rdf-sparql-protocol/.
7. DARPA. DAML. http://www.daml.org/, 3 February 2007.
8. 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/.
9. Leigh Dodds. XML Army Knife. http://xmlarmyknife.org/api/rdf/sparql/query, 5 December 2006.
10. 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.
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. Jena. A Semantic Web Framework for Java. http://jena.sourceforge.net/, 30 November 2006.
13. Holger Knublauch, Mark A. Musen, and Alan L. Rector. Editing description logic ontologies with the protégé owl plugin. In Volker Haarslev and Ralf Möller, editors, *Description Logics*, volume 104 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2004.
14. 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.
15. Nuno Lopes, Cláudio Fernandes, and Salvador Abreu. Representing and querying multiple ontologies with contextual logic programming. July 2008.
16. Frank Manola and Eric Miller. RDF Primer. W3C Recommendation, World Wide Web Consortium, February 2004.
17. Deborah L. McGuinness and Frank van Harmelen. OWL Web Ontology Language Overview. W3C Recommendation, World Wide Web Consortium, February 2004.
18. Protégé. Free, open source ontology editor and knowledge-based framework. http://protege.stanford.edu/, 30 November 2006.
19. Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. Technical report, W3C, 2006. Available at: http://www.w3.org/TR/2006/CR-rdf-sparql-query-20060406/.
20. Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical owl-dl reasoner. *J. Web Sem.*, 5(2):51–53, 2007.
21. Software Systems Institute. Racer Manager. http://racerproject.sourceforge.net/, 19 February 2007.

22. Vangelis Vassiliadis. Thea OWL Parser for Prolog. http://www.semanticweb.gr/TheaOWLParser/, 12 October 2006.
23. Vangelis Vassiliadis. *Thea A Web Ontology Language - OWL Parser for [SWI] Prolog.* http://www.semanticweb.gr/downloads/Thea%20OWL%20Parser.doc, 19 January 2007.
24. W3C. Wine Ontology. http://www.w3.org/TR/owl-guide/wine.rdf, 22 July 2006.