Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 200 (2008) 67–85
www.elsevier.com/locate/entcs

An RDF Query Language based on Logic Programming

Jesús M. Almendros-Jiménez[1,2]
Dpto. de Lenguajes y Computación. Universidad de Almería.

Abstract

In this paper we investigate an extension of XQuery for querying (and inferring) from RDF documents. Following a graph based approach for specifying queries against RDF, XQuery is extended with construction of answers and boolean predicates for RDF entailment relationship inference. We will also study how to implement it in logic programming by using logic rules for executing RDF/XQuery queries.

Keywords: XML, RDF, XQuery, Logic Programming

1 Introduction

According to the *Semantic Web* requirements [6], Web data needs to be enriched with meaning from new formalisms for expressing knowledge about the domain of interest. It has lead to the definition of new languages for representing domain/data hierarchies together with mechanisms for expressing relationships between data and domains. More sophisticated languages are based on the logic as a way to enrich the formal modeling of the knowledge. In this framework, Web data represented by HMTL/XML has to be enriched by meta-data in which modeling is mainly achieved by means of the *Resource Description Framework (RDF)* [25] and the *Ontology Web Language (OWL)* [23]. RDF and OWL can be also used for expressing both data and meta data.

RDF is a way for data and meta data representation. RDF *statements* consists of triples $(Subject, Property, Object)$. RDF data model is a *directed graph* whose nodes are the subjects and objects and whose arcs are the properties. Nodes are labeled by means of *URIs* describing *resources* or *literals* (i.e. strings or numbers) or are unlabeled, called *blank nodes*. Blank nodes are usually used to group properties. Edges are always labeled by URIs representing a relationship between the subject and the object. *RDF Schema (RDFS)* [24] extends RDF by defining the schema of RDF statements, as a way for creating application specific vocabularies. This schema represents an *ontology* in which relationships between RDF items can be specified. RDF together with RDFS allow the description of knowledge about a specific domain together with a set of instances of such domain. From RDFS,

[1] This work has been partially supported by the EU (FEDER) and the Spanish MEC under grant TIN2005-09207-C03-02.

[2] Email: jalmen@ual.es

doi:10.1016/j.entcs.2008.04.093

entailment relations can be defined, including the transitive closure of RDF statements.

XQuery [27,8] is a typed functional language devoted to express queries against XML documents. It contains *XPath 2.0* [26] as a sub-language. *XPath 2.0* supports navigation, selection and extraction of fragments from XML documents. *XQuery* also includes expressions to construct new XML values and to join multiple documents.

In recent papers [2,3,1], we have proposed a logic programming based implementation of the *XPath* and *XQuery* languages. Such implementation allows to express *XPath* and *XQuery* queries in logic programming. With this aim *XML* documents are translated into a logic program by means of facts and rules, and an *XPath*/*XQuery* query is executed by specializing the logic program representing the input XML document and generating one or more specific goals for the query. From the computed answers for the goals we are able to rebuild the output XML document.

The aim of our approach, called *XIndalog*, is to have a logic programming based implementation of *XPath* and *XQuery*, in order to be combined with the inference capabilities of logic programming for the Semantic Web. It has motivated the introduction of RDF in our framework. In this proposal, logic programming can also be used for combining and querying data from *heterogeneous resources* in which some of them offers data in XML format and others as RDF statements. Logic programming is a suitable framework for such combination once the inference capabilities of logic programs can be useful for inferring new semantic information (for instance, RDFS entailment) from RDF resources.

In this paper we present an extension of *XQuery* for the querying of RDF documents, and we study how to implement it in logic programming. Such extension and implementation can be summarized as follows:

- The extension allows the *handling of RDF graphs* in *XQuery* expressions. Such *XQuery* expressions can also combine *XPath* expressions for the handling of XML data.
- It allows the use of *built-in predicates* for the inference of RDF relationships to be used in the queries. Such RDF relationships includes the transitive closure of the subclass relationships and, in general, complex *join operations between RDF triples*.
- It allows the querying of RDF/XML documents but also the *construction of the answer* by means of an XML document. In particular the answer could represent the *serialized version* of a RDF document. Therefore we are able to work with *XML/RDF* documents as input and as output.
- In addition, we will present how to *implement such extension of XQuery in logic programming*. With this aim we have to *represent RDF documents in logic programs*, and we have to define *logic rules for inferring information* from RDF statements. Finally, we have to describe how to *translate XQuery expressions involving RDF queries into logic programming*.

In the last years a great effort has been made for defining query languages for RDF documents (see [4,11] for surveys about this topic). The proposals mainly fall on extensions of *SQL*-style syntax for handling the *graph based RDF structure*. In this line the most representative languages are SquishQL [15], SPARQL [16] and RQL [12]. Moreover, there are some languages based on extensions of *XPath*, *XSLT* and *XQuery* languages – the W3C consortium proposals for query languages for the Semantic Web–. In this line the most representative languages are XQuery for RDF (the Syntactic Web Approach) [18], RDF Twig [28], RDFPath [21], RDFT [9] and XsRQL [13]. Finally, some of them are logic-based languages, therefore *rule based languages*. This is the case of TRIPLE [20], N3QL [5] and XCerpt [19,10]. They have their own syntax similar to *deductive logic languages*.

XQuery can be adapted to the handling of *RDF* documents by means of some kind of *serialization* of *RDF* documents in *XML*. Such serialization allows queries on RDF documents to be expressed by means of (extensions of) *XPath*. This serialization has been followed in previous proposals [18,28,9,21,13] on extensions of *XPath*, *XQuery* and *XSLT* for RDF. However, in our opinion, we can define an extension of *XQuery* by using *triple based syntax* for representing RDF queries, similarly to SQL-style proposals for RDF [15,16,12]. The advantage of such triple based syntax is that queries do not depend on the selected serialization. Moreover, in most of proposals serialization makes queries are too sophisticated.

In addition, we can extend *XQuery* with reasoning/inferring capabilities on RDF, in order to handle semantic information. With this aim we will introduce built-in predicates for RDF/RDFS properties like `rdf:type`, `rdfs:domain`, `rdfs:range`, `rdfs:subClassOf`, `rdfs:subPropertyOf`, and so on. In most of the cited RDF query languages the inferring of new semantic information from RDF, that is, the computation of the RDFS entailment relation, is achieved by means of functions/operators for the traversal of the RDF graph. Fortunately, logic programming is a suitable tool for supporting inference, and logic rules can be used for computing the entailment relationship.

Once we have extended *XQuery* with RDF triples and reasoning capabilities, queries can make joins between RDF and XML documents and therefore *XQuery* is able to handle heterogeneous data resources. This combination is only followed by some proposals –for instance [10,18,14,17]–. In addition, our proposed language can be fully implemented in logic programming and therefore it could be also extended in the future with more powerful reasoning capabilities like with OWL restricting ontologies to be expressed in logic programming in the line of *Description Logic Programs (DLP)* [29].

On the other hand, one of the advantages of our proposal is that XML/RDF documents can work as input and as output. Some proposals about RDF query languages lack on the construction of the output as new RDF triples. However, our approach generates XML documents as output of RDF queries and therefore it allows the composition of queries. *XQuery* allows to specify the XML format of the output document and therefore the output can be expressed as an XML document and also as a serialization of RDF.

Finally, we will describe how to implement such extension of *XQuery* in logic programming. We have described in our previous works [2,3] how to implement *XPath* and *XQuery* in logic programming. Therefore we have now to describe how the extension to RDF is achieved.

Firstly, RDF documents can be represented by means of facts. We follow a different approach to [7], because our formalism represents triples with a predicate called `triple` and a fact for each triple *(Subject,Property,Object)* of RDF. However, the representation contains only the basic triples and therefore specific rules has to be defined for those entailed by the RDFS semantics.

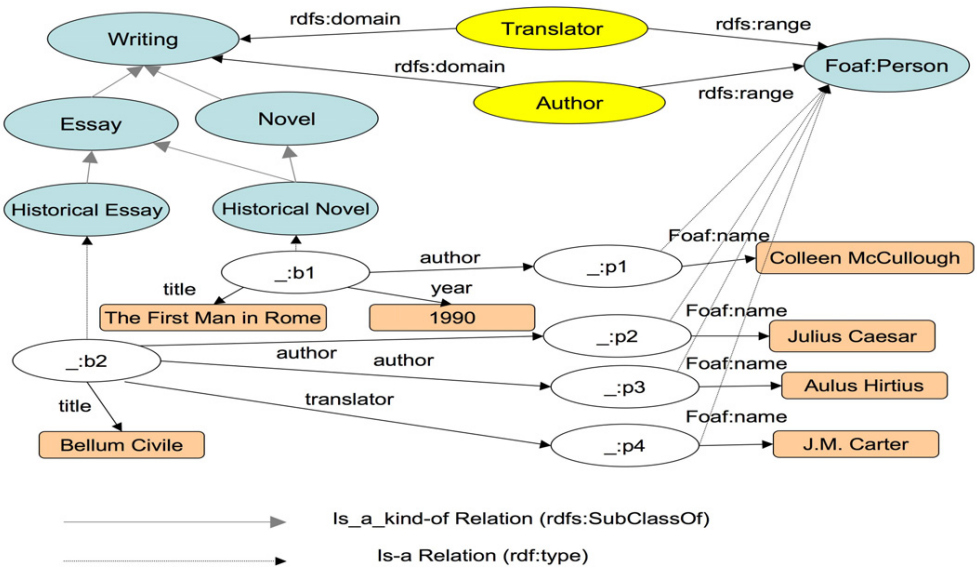
Secondly, some rules are introduced in order to know which is the `rdfs:domain/rdfs:range` of a given property and to compute the `rdfs:subClassOf / rdfs:subPropertyOf` relationships. Such relationships can be used in *XQuery* expressions by means of the corresponding built-in predicates.

The structure of the paper is as follows. Section 2 will present the translation of RDF documents into Prolog and will describe the reasoning capabilities by means of logic rules; Section 3 will present the extension of *XQuery* to cover with RDF triples; Section 4 will present the translation into logic programming; and finally, Section 5 will conclude and present future work.

2 RDF Documents into Logic Programming

In this section we show how to represent RDF documents in logic programming. This is the basis of our extension of *XQuery*.

Fig. 1. RDF example



In [7] a proposal for representation of RDF statements has been given. However, in our approach we adopt a different and more direct representation of triples as (Prolog) facts of the form $triple(Subject, Property, Object)$. For instance, w.r.t. the running example in Figure 1 borrowed from [4], assuming it is stored in 'http://www.example.org/books' we will consider facts:

```
triple(_:b1,rdf:type,'Historical Novel','http://www.example.org/books',1).
triple(_:b1,books:author,_:p1,'http://www.example.org/books',2).
triple(_:p1,foaf:name,'Colleen McCullough','http://www.example.org/books',3).
triple('Historical Novel',rdfs:subClassOf,'Novel','http://www.example.org/books',4).
triple(translator,rdfs:domain,'Writing','http://www.example.org/books',5).
triple(translator,rdfs:range,foaf:Person,'http://www.example.org/books',6).
```

Each fact for `triple` represents each triple of RDF. In addition, we have to number each triple (according to the RDF semantics there is no order between RDF triples and therefore any numbering identifying each triple is enough). The use of the numbering will be explained later. With this representation we can write logic predicates for computing RDF(S) relationships. For instance, the transitivity of the `rdfs:subClassOf` relationship can be computed as follows:

```
rdfs_subClassOf(SubClass,Class,Res):- triple(SubClass,rdfs:subClassOf,Class,Res,N).
rdfs_subClassOf(SubClass,Class,Res):- triple(SubClass,rdfs:subClassOf,ClassAux,Res,N),
            rdfs_subClassOf(ClassAux,Class,Res).
```

For instance, w.r.t. the running example, the goal $:-$`rdfs_subClassOf`$('$`Historical Essay`$','$`Writing`$', '$`http://www.example.org/books`$')$ is successful. Similar predicates can be defined for others RDFS and RDF relationships, for instance a more complex one is:

```
rdfs_domain(Resource,Class,Res):-triple(Resource,rdfs:domain,Class,Res,N).
rdfs_domain(Resource,Class,Res):-rdfs_subClassOf(Class,ClassAux,Res),
              rdfs_domain(Resource,ClassAux,Res).
```

From which we can compute the domain of a given resource: for instance `books:author` has a domain `'Historical Novel'` because the goal $:-$`rdfs_domain(books:author,'Historical Novel','http://www.example.org/books')` is successful.

In this way, logic programming can be used as reasoning machine for most usual RDFS entailment relationships. Such relationships can be computed from RDF and RDFS properties, and also can involve meta-data relationships. Some of them can required to make joins between RDF(S) triples. We can define a rich set of built-in predicates in a similar way of other RDF query languages like RQL [12]. Among others we can define: `rdfs_subPropertyOf`, `leafClass`, `leafProperty`, `nca` (nearest common ancestor of two classes), `topClass`, `topProperty`, etc. In addition, we can define built-in typing predicates like `is_class`, `is_property` and `is_type`. Now, we would like to present our extension of *XQuery* which handles RDF triples and includes the built-in predicates defined in this way.

3 An Extended XQuery for RDF

The proposed extension to *XQuery* has to extend the syntax for the traversal of RDF triples. We have adopted a simple syntax in which **for-let-where-return** expressions allow to traverse RDF triples by means of the **for** expression. The following table shows the syntax of the extended *XQuery* language, representing the core of the language.

Core XQuery

xquery:= **namespace** *name* **:** *resource* **in** *xquery*
| *dexpr* | $< tag\ att = vexpr, \ldots, att = vexpr >' \{' xquery, \ldots, xquery' \}' < /tag >$
| *flwr* | *value*.

dexpr:= **document**(*doc*) *'/'* *expr*.
rdfdoc := **rdfdocument**(*doc*).
flwr:= **for** *$var* **in** *vexpr* [**where** *constraint*] **return** *xqvar*
| **for** *($var,$var,$var)* **in** *rdfdoc* [**where** *constraint*] **return** *xqvar*
| **let** *$var* **:=** *vexpr* [**where** *constraint*] **return** *xqvar*.

xqvar:= *vexpr* | $< tag\ att = vexpr, \ldots, att = vexpr >' \{' xqvar, \ldots, xqvar' \}' < /tag >$
| *flwr* | *value*.
vexpr:= *$var* | *$var '/' expr* | *dexpr* | *value*.
expr:= *text()* | *tag-name* | *tag-name[expr]* | *'/' expr*.
constraint := $Op(vexpr, \ldots, vexpr)$ | *constraint* **'or'** *constraint* | *constraint* **'and'** *constraint*.

where "name **:** resource" assigns name spaces to URL resources; "value" can be URL/URI's, name spaces, strings, numbers or XML documents; tag's are XML labels; att's are attribute names; doc's are URL's; and finally, *Op*'s can be selected from the usual binary operators: <=, >= < >, =, =/=, and RDF built-in predicates.

Basically, a simple *XQuery* language has been extended with two constructions: the **namespace** statement allowing the declaration of URIs taken from other resources –some queries can make use of the name space–; and a new **for** expression for traversing triples from a RDF document whose location is specified by means of **rdfdocument** primitive. In addition, the **where** construction includes boolean conditions of the form $rdf\_pred(vexpr, \ldots, vexpr)$ which can be used for testing RDF/RDFS properties (for instance *rdf_pred* can be one of `rdfs:domain`, `rdfs:range`, `rdfs:subClassOf`), including not only binary but N-ary built-in predicates (for instance, `nca`, the nearest common ancestor).

The above *XQuery* is a typed language in which there are two kinds of variables: those variables used in *XPath* expressions, and those used in RDF triples. However they can be compared by means of boolean expressions, and they can be used together for the construction of the answer.

We have restricted ourselves to a simple *XQuery* language in which other built-in constructions for *XPath* can be added, and also it can be enriched with other *XQuery* constructions, following the W3C recommendations [27]. However, now we will show that with this small extension of *XQuery* we are able to express some interesting queries.

Now, we would like to show some examples of our language. We will take as query examples those proposed in [4].

3.1 Selection and Extraction Queries

Query 1: The first query to be expressed in our language is *"Select all Essays together with their authors"*:

```
<essays>{
for ($Book, $BookProperty, $Author) in rdfdocument('http://www.example.org/books')
return
for ($Bookauthor, $AuthorProperty, $Name) in rdfdocument('http://www.example.org/books')
where $Author=$Bookauthor and $BookProperty=books:author and $AuthorProperty=foaf:name
        and rdf:type($Book,books:Essay)
return <essay>{
              <book> $Book < / book>
              <authorname> $Name < / authorname>
       } </ essay>
} </ essays>
```

In this example we can see how triples are traversed by means of the **for** expression in which variables are used for representing each element of the triple. The **where** expression is used to make a join between the tuples and after the answer is built in the **return** expression, generating an XML document. In addition, a binary built-in predicate called `rdf : type` is called to reason from the RDF schema –it checks whether the given book is of type *Essay*–.

Query 2: The second query is: *"Select all data items with any relation to the book titled 'Bellum Civile'"*. This is a structural query in which a subset of the RDF is required: the subset related with the book titled 'Bellum Civile'. In this case an extensive use of RDF predicates is required in our approach, but it can be expressed as follows:

```
<related> {
for ($Book, $TitleProperty, $Title) in rdfdocument('http://www.example.org/books')
return
for ($Subject, $Property, $Object) in rdfdocument('http://www.example.org/books')
where $TitleProperty=books:title
        and $Title='Bellum Civile' and ($Property=rdf:type and rdf:type($Book,$Object))
        or ($Property=books:author and $Subject=$Book) or ...
return <item> {
              <subject> $Subject < / subject>
              <property> $Property < / property>
              <object> $Object < / object>
       } </ item>
} </ related>
```

Query 3: The third query proposed in [4] is: *"Select all data items except ontology information and translators"*. In this case we can formulate the query as **Query 2**, but the query is more concise using negation on boolean conditions:

```
<related> {
for ($Subject, $Property, $Object) in rdfdocument('http://www.example.org/books')
where $Property=/=rdfs:subClassof and $Property=/=rdf:type and $Property=/=rdfs:domain
and $Property=/=rdfs:range and $Property=/=books:translator
return <item> {
              <subject> $Subject < / subject>
              <property> $Property < / property>
              <object> $Object < / object>
       } </ item>
} </ related>
```

3.2 Restructuring Queries

Query 4: Now, the query is: *"Invert the relation author (from a book to an author) into a relation authored (from an author to a book)"*. This kind of queries are possible in our approach since we allow the construction of an answer. Let us suppose that we serialize RDF documents as follows in order to build the answer:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#",
xmlns:books="http://example.org/books#"> {
for ($Book, $BookProperty, $Author) in rdfdocument('http://www.example.org/books')
where $BookProperty=books:author
return <rdf:Description about=$Author> {
              <books:authored> $Book < / books:authored>
       } </ rdf:Description>
} </ rdf:RDF>
```

3.3 Aggregation Queries

The proposed queries **Query 5**: *"Return the last year in which an author with name 'Julius Caesar' published something"* and **Query 6**: *"Return each of the subclasses of 'Writing', together with the average numbers of authors per publication of that subclass"* are aggregation queries and we have not considered aggregation operators in our core *XQuery* language. If we consider them, we would have to use aggregation operators in logic programming, by using `findall` predicate together with some specific rules: `average`, `max`, etc. This is still out of the scope of our proposal.

3.4 Combination and Inference Queries

Query 7: This query expresses: *"Combine the information about the book titled 'Civil War' and authored by 'Julius Cesar' with the information about the book with identifier 'Bellum Civile'"*. This query basically expresses that some books can have different titles but corresponding to the same authors, and therefore new RDF triples can be inferred. It can be expressed in general as follows:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#",
xmlns:books="http://example.org/books#"> {
for ($Book1, $BookProperty1, $Title) in rdfdocument('http://www.example.org/books')
return
for ($Book2, $BookProperty2, $Author) in rdfdocument('http://www.example.org/books')
where $Book1=$Book2 and $BookProperty1=books:title and $BookProperty2=books:author
       return <rdf:Description about=$Book1> {
             <books:title > $Title < / books:title>
             <books:author> $Author < / books:author>
      } </ rdf:Description>
} </ rdf:RDF>
```

Query 8: This query is: *"Returns the transitive closure of the* `subClassOf` *relation"*. It can be expressed as follows:

```
<subclassof> {
for ($Subject1, $Property1, $Object1) in rdfdocument('http://www.example.org/books')
where $Property1=rdf:type and $Object1=rdfs:Class
return
<class classname=$Subject1> {
for ($Subject2, $Property2, $Object2) in rdfdocument('http://www.example.org/books')
where $Property2=rdf:type and $Object2=rdfs:Class
       and rdfs:subClassof($Subject2,$Subject1)
       return <subclass> {
              $Subject2
              } </subclass>
} </class>
} </subclassof>
```

Let us remark that we have decided to make a nested query in this case for grouping all subclasses together with the corresponding superclass.

Query 9: Finally, the query *"Return the co-author relation between two persons that stand in author relationships with the same book"* is as follows:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#",
xmlns:books="http://example.org/books#"> {
for ($Book1, $BookProperty1, $Author1) in rdfdocument('http://www.example.org/books')
return
for ($Book2, $BookProperty2, $Author2) in rdfdocument('http://www.example.org/books')
where $Book1=$Book2 and $BookProperty1=books:author and $BookProperty2=books:author
       and $Author1 =/= $Author2
return <rdf:Description about=$Author1> {
             <books:co-author> $Author2 < / books:co-author>
      } </ rdf:Description>
} </ rdf:RDF>
```

4 Extended XQuery into Logic Programming

Each core *XQuery* expression can be translated into a set of rules and specific goals in logic programming. We have studied in our previous works how to translate

XPath [3] and *(Non-RDF) XQuery* [2]. However the extension of *XQuery* to cover with RDF triples needs a specific translation into logic rules.

In [3,2] we have shown how to translate XML documents into logic programming by means of logic rules representing the *inner nodes* of the XML document (called *schema rules*) and facts representing the *leaves* of the XML document. The translation of *XPath* and *(Non-RDF) XQuery* into logic programming consists of the specialization of the schema rules w.r.t. the given query, and the generation of (one or more) specific goals from the given query.

In order to extend our translation to cover with RDF, we have to combine such *XPath/(Non-RDF) XQuery* rules with the *RDF* rules presented in Section 2, and calls to the `triple` facts. However, we can still make a simple and self-content presentation of the translation by restricting ourselves to the RDF fragment of *XQuery* expressed by the following rules:

RDF fragment of XQuery

xquery:= **namespace** *name* **:** *rdfdoc* **in** *xquery*
| $< tag\ att = vexpr, \ldots, att = vexpr >' \{'xquery, \ldots, xquery'\}' < /tag >$
| *flwr* | *value.*
rdfdoc := rdfdocument(doc).

flwr:= **for** *($var,$var,$var)* **in** *rdfdoc [***where** *constraint]* **return** *xqvar.*
xqvar:= *vexpr* | $< tag\ att = vexpr, \ldots, att = vexpr >' \{'xqvar, \ldots, xqvar'\}' < /tag >$
| *flwr* | *value.*
vexpr:= $var | *value.*
constraint := *Op(vexpr, . . . , vexpr)* | *constraint* **'or'** *constraint* | *constraint* **'and'** *constraint.*

Let us remark that previous queries have been expressed by using the previous syntax. Now, let us proceed with a key point of our translation: the translation of XML documents into a logic program, and analogously, the reconstruction from a logic program of an XML document. With this aim, the following section will show the proposed translation of XML documents in logic programing in the quoted papers [3,2].

4.1 Translating XML Documents into Logic Programming

In order to define our translation we need to number the nodes of the XML document. A similar numbering of XML documents has been already adopted in some proposals for representing XML in relational databases [22].

Given an XML document, we can consider a new XML document called *node-numbered XML document* as follows.

Starting from the root element numbered as 1, the node-numbered XML document is numbered using an attribute called **nodenumber** [3] where each j-th child of a tagged element is numbered with the sequence of natural numbers $i_1 \ldots . i_t.j$ whenever the parent is numbered as $i_1 \ldots . i_t$: $< tag\ att_1 = v_1, \ldots, att_n = v_n, \mathbf{nodenumber} = \mathbf{i_1} \ldots . \mathbf{i_t.j} > elem_1, \ldots, elem_s < /tag >$. This is the case of tagged elements. If the j-th child is of a basic type (non tagged) and the

[3] It is supposed that "nodenumber" is not already used as attribute in the tags of the original XML document.

parent is an inner node, then the element is labeled and numbered as follows: $< unlabeled$ $\mathbf{nodenumber = i_1.\ldots.i_t.j} > elem < /unlabeled >$; otherwise the element is not numbered. It gives to us a *hierarchical and left-to-right numbering* of the nodes of an XML document.

An element in an XML document is further left in the XML tree than another when the node number is smaller w.r.t. the lexicographic order of sequences of natural numbers. Any numbering that identifies each inner node and leaf could be adapted to our translation.

In addition, we have to consider a new document called *type and node-numbered XML document* numbered using an attribute called **typenumber** as follows. Starting the numbering from 1 in the root of the node-numbered XML document, each tagged element is numbered as: $< tag\ att_1 = v_1, \ldots, att_n = v_n, nodenumber = i_1\ldots., i_t.j, \mathbf{typenumber = k} > elem_1, \ldots, elem_s < /tag >$. The type number k of the tag is equal to $l + n + 1$ whenever the type number of the parent is l, and n is the number of tagged elements weakly distinct [4] occurring in leftmost positions at the same level of the XML tree [5].

Now, the translation of the XML document into a logic program is as follows. For each inner node in the type and node numbered XML document $< tag\ att_1 = v_1, \ldots, att_n = v_n, nodenumber = i, typenumber = k > elem_1, \ldots, elem_s < /tag >$ we consider the following rule, called *schema rule*:

$$
\begin{array}{l}
tag(tagtype(Tag_{i_1}, \ldots, Tag_{i_t}, [Att_1, \ldots, Att_n]), NTag, k, Doc) \text{:-} \\
\qquad tag_{i_1}(Tag_{i_1}, [NTag_{i_1}|NTag], r, Doc), \ldots, \\
\qquad tag_{i_t}(Tag_{i_t}, [NTag_{i_t}|NTag], r, Doc), \\
\qquad att_1(Att_1, NTag, r, Doc), \ldots, \\
\qquad att_n(Att_n, NTag, r, Doc).
\end{array}
$$

where *tagtype* is a new function symbol used for building a Prolog term containing the XML document; $\{tag_{i_1}, \ldots, tag_{i_t}\}$, $i_j \in \{1, \ldots, s\}$, $1 \leq j \leq t$, is the *set of tags* of the tagged elements $elem_1, \ldots, elem_s$; $Tag_{i_1}, \ldots, Tag_{i_t}$ are variables; $att_1, \ldots, att_n$ are the attribute names; $Att_1, \ldots, Att_n$ are variables, one for each attribute name; $NTag_{i_1}, \ldots, NTag_{i_t}$ are variables (used for representing the last number of the node number of the children); $NTag$ is a variable (used for representing the node number of *tag*); k is the type number of *tag*; and finally, r is the type number of the tagged elements $elem_1, \ldots, elem_s$ [6].

In addition, we consider facts of the form: $att_j(v_j, i, k, doc)$ $(1 \leq j \leq n)$, where *doc* is the name of the document. Finally, for each leaf in the type and node numbered XML document: $< tag\ nodenumber = i, typenumber = k > value < /tag >$, we consider the *fact*: $tag(value, i, k, doc)$. For instance, let us consider the following XML document called "books.xml":

[4] Two elements are weakly distinct whenever they have the same tag but not the same structure.

[5] In other words, type numbering is done by levels and in left-to-right order, but each occurrence of weakly distinct elements increases the numbering in one unit.

[6] Let us remark that since *tag* is a tagged element, then $elem_1, \ldots, elem_s$ have been tagged with "unlabeled" labels in the type and node numbered XML document when they were not labeled; thus they must have a type number.

```
<books>
<book year="2003">
    <author>Abiteboul</author>
    <author>Buneman</author>
    <author>Suciu</author>
    <title>Data on the Web</title>
    <review>A <em>fine</em> book.</review>
</book>
<book year="2002">
    <author>Buneman</author>
    <title>XML in Scotland</title>
    <review><em>The <em>best</em> ever!</em></review>
</book> </books>
```

Now, the previous XML document can be represented by means of a logic program as follows:

Rules (Schema):	*Facts (Document):*
books(bookstype(Book, []), NBooks,1,Doc) :-	*year('2003', [1, 1], 3,"books.xml").*
book(Book, [NBook|NBooks],2,Doc).	*author('Abiteboul', [1, 1, 1], 3,"books.xml").*
book(booktype(Author, Title, Review, [Year]),	*author('Buneman', [2,1, 1], 3,"books.xml").*
NBook ,2,Doc) :-	*author('Suciu', [3,1,1], 3,"books.xml").*
author(Author, [NAu|NBook],3,Doc),	*title('Data on the Web', [4, 1, 1], 3,"books.xml").*
title(Title, [NTitle|NBook],3,Doc),	*unlabeled('A', [1, 5, 1, 1], 4,"books.xml").*
review(Review, [NRe|NBook],3,Doc),	*em('fine', [2, 5, 1, 1], 4,"books.xml").*
year(Year, NBook,3,Doc).	*unlabeled('book.', [3, 5, 1, 1], 4,"books.xml").*
review(reviewtype(Un,Em,[]),NReview,3,Doc):-	
unlabeled(Un,[NUn|NReview],4,Doc),	*year('2002', [2, 1], 3,"books.xml").*
em(Em,[NEm|NReview],4,Doc).	*author('Buneman', [1, 2, 1], 3,"books.xml").*
review(reviewtype(Em,[]),NReview,3,Doc):-	*title('XML in Scotland', [2, 2, 1], 3,"books.xml").*
em(Em,[NEm|NReview],5,Doc).	*unlabeled('The', [1, 1, 3, 2, 1], 6,"books.xml").*
em(emtype(Unlabeled,Em,[]),NEms,5,Doc) :-	*em('best', [2, 1, 3, 2, 1], 6,"books.xml").*
unlabeled(Unlabeled,[NUn|NEms],6,Doc),	*unlabeled('ever!', [3, 1, 3, 2, 1], 6,"books.xml").*
em(Em, [NEm|NEms],6,Doc).	

Here we can see the translation of each tag into a predicate name: *books*, *book*, etc. Each predicate has four arguments, the first one, used for representing the XML document structure, is encapsulated into a function symbol with the same name as the tag adding the suffix *type*. Therefore, we have *bookstype*, *booktype*, etc. The second argument is used for numbering each node –in reverse order due to the use of Prolog lists–; the third argument of the predicates is used for numbering each type; and the last argument represents the document name. The key element of our translation is to be able to recover the original XML document from the set of rules and facts.

The previous translation has the following peculiarities. In order to specify the order in an XML document each fact is numbered from left to right and by levels in the XML tree. In addition, the hierarchical structure of the XML records is expressed by means of the identifier of each record (the number of the parent) and the length of the number (the children has a larger number). The type number

makes possible to distinguish which schema rule is appliable for records with the same tag but different structure.

4.2 Translation of the RDF Fragment of XQuery into Logic Programming

Let us suppose the **Query 1** of our extended *XQuery*:

```
<essays> {
namespace books :  http://example.org/books# in
namespace rdf :  http://www.w3.org/1999/02/22-rdf-syntax-ns# in
for ($Book, $BookProperty, $Author) in rdfdocument('http://www.example.org/books')
return
for ($Bookauthor, $AuthorProperty, $Name) in rdfdocument('http://www.example.org/books')
where $Author=$Bookauthor and $BookProperty=books:author and $AuthorProperty=foaf:name
       and rdf:type($Book,books:Essay)
return <essay> {
             <book> $Book < / book>
             <authorname> $Name < / authorname>
      } </ essay>
} </ essays>
```

According with the translation of RDF into logic programming proposed in Section 2, and the translation of XML documents into logic programming proposed in Section 4.1, we can consider the following rules:

```
essays(essaystype(Essay,[]),Node,1,"result.xml"):-
        essay(Essay,[NodeEssay|Node],2,"result.xml").
essay(essaytype(booktype(Book,[]),authornametype(Name,[]),[]),Node,2,"result.xml"):-
        join(Book,Name,Node,2).
join(Book,Name,[M,1],2):-
      triple(Book,BookProperty,Author, 'http://www.example.org/books',N),
      triple(Bookauthor,AuthorProperty,Name,'http://www.example.org/books', M),
       eq(Author,Bookauthor),
       eq(BookProperty,books:author),
       eq(AuthorProperty,foaf:name),
       rdf:type(Book,books:Essay,'http://www.example.org/books').
```

The translation has three rules. The first rule describes the schema rule for the output document (i.e. *essays* are composed by *essay* items). It is obtained from the **return** expression.

The second rule (i.e. the *essay* rule) is not a schema rule. It is used for computing "facts" of the output XML document. Such facts for output documents have not the same form as the facts generated from input XML documents. They are facts of the form:

```
essay(essaytype(booktype(_:b1,[]), authornametype('Colleen McCullough',[]),[]), [3,1],2)
```

in which a Prolog term "$essaytype(booktype(\_:b1,[]), authornametype('Colleen\ McCullough',[]),[])$" has been built. This Prolog term represents a fragment of the output XML document, and it has been built from the *join* between the following

RDF triples.

```
triple(_:b1,books:author,_:p1,2).
triple(_:p1,foaf:name,'Colleen McCullough',3).
```

Let us remark that such fragments are the same kind of fragments generated from the schema rules in our representation for input XML documents. From such fragments, the output XML document can be rebuilt by using the type and node numbering, and the schema rules. For instance, from the above fact and the schema rule *Essays*, we are able to rebuild:

$$
\begin{array}{l}
< essays >< essay > \\
< book > \_ : b1 < /book > \\
< authorname >' ColleenMcCullough' < /authorname > \\
< /essay >< /essays >
\end{array}
$$

which is one of the records of the output XML document. The third rule (i.e. *join*) makes the join between RDF triples in **for** expressions, and uses the translation of the built-in predicates for checking RDF properties on **where** expressions. In addition, the equalities and inequalities $=, >, <, = / =$, etc occurring in the **where** expression are translated into logic predicates *eq*, *gth*, *lth*, *neq*, etc.

Now, we have to consider a goal for the retrieving of the answer. Such goal is built from the main tag of the **return** expression (those involving the *join*). In this example, the goal is $: -essay(Essay, Node, Type, Doc)$ obtaining as answers:

```
(1) Essay= essaytype(booktype(_:b1,[]),authornametype('Colleen McCullough',[]),[]),[]), Node=[3,1],Type=1
(2) Essay= essaytype(booktype(_:b2,[]),authornametype('Julius Cesar',[]),[]),[]), Node=[8,1],Type=1
(3) Essay= essaytype(booktype(_:b2,[]),authornametype('Aulus Hirtius',[]),[]),[]), Node=[9,1],Type=1
(4) Essay= essaytype(booktype(_:b2,[]),authornametype('J.M.Carter',[]),[]),[]), Node=[10,1],Type=1
```

assuming `triple` includes the facts:

```
triple(_:b1,books:author,_:p1,2).
triple(_:p1,foaf:name,'Colleen McCullough',3).
triple(_:b2,books:author,_:p2,5).
triple(_:b2,books:author,_:p3,6).
triple(_:b2,books:author,_:p4,7).
triple(_:p2,foaf:name,'Julius Cesar',8).
triple(_:p3,foaf:name,'Aulus Hirtius',9).
triple(_:p4,foaf:name,'J.M.Carter',10).
```

This set of computed answers represents the following goal instances:

```
(1) essay(essaytype(booktype(_:b1,[]),authornametype('Colleen McCullough',[]),[]),[]),[3,1],1).
(2) essay(essaytype(booktype(_:b2,[]),authornametype('Julius Cesar',[]),[]),[]),[8,1],1).
(3) essay(essaytype(booktype(_:b2,[]),authornametype('Aulus Hirtius',[]),[]),[]),[9,1],1).
(4) essay(essaytype(booktype(_:b2,[]),authornametype('J.M.Carter',[]),[]),[]),[10,1],1).
```

and the output XML document (i.e. "result.xml") can be built from this set of facts and the schema rule:

```
essays(essaystype(Essay,[]),Node,1,"result.xml"):-
        essay(Essay,[NodeEssay|Node],2,"result.xml").
```

as follows:

```
< essays >
< essay >< book > _ : b1 < /book >< authorname > Colleen McCullough < /authorname >< /essay >
< essay >< book > _ : b2 < /book >< authorname > Julius Cesar < /authorname >< /essay >
< essay >< book > _ : b2 < /book >< authorname > Aulus Hirtius < /authorname >< /essay >
< essay >< book > _ : b2 < /book >< authorname > J.M.Carter < /authorname >< /essay >
< /essays >
```

Let us remark that the output XML document is built using the node and type numbering in which the parent is numbered with a larger number than children. In the example, the *essays* label is numbered as *[1]* and the children as *[3,1]*, *[8,1]*, *[9,1]*, *[10,1]*. With this aim we have numbered RDF triples in Prolog facts.
Let us see the case of **Query 3**. The translation is similar to the **Query 1**, where we use the predicate *neq* to represent the =/= boolean operator in *XQuery*.

```
<related>
{
for ($Subject, $Property, $Object) in rdfdocument('http://www.example.org/books')
where $Property=/=rdfs:subClassof and $Property=/=rdf:type and $Property=/=rdfs:domain
and $Property=/=rdfs:range and $Property=/=books:translator
return <item>
       {
              <subject> $Subject < / subject>
              <property> $Property < / property>
              <object> $Object < / object>
       }
       </ item>
}
</ related>
```

In this case the goal is *:-item(Item,Node,Number,Doc)* and the translation is:

```
related(relatedtype(Item,[]),Node,1,"result.xml"):-
        item(Item,[NodeItem|Node],2,"result.xml").
item(itemtype(subjecttype(Subject,[]),propertytype(Property,[]),
         objecttype(Object,[]),[]),Node,2,"result.xml"):-
        join(Subject,Property,Node,2).
join(Subject,Property,[N,1],2):-
        triple(Subject,Property,Object,'http://www.example.org/books',N),
        neq(Property,rdfs:subClassof),
        neq(Property,rdf:type),
        neq(Property,rdfs:domain),
        neq(Property,rdfs:range),
        neq(Property,books:translator).
```

In the case of **Query 4** two attributes have been added to the output document: the namespaces for rdf and books. They are represented as attributes of the schema rules and facts of the output document, following the criteria of the translation presented in Section 4.1.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#",
xmlns:books="http://example.org/books#">
{
for ($Book, $BookProperty, $Author) in rdfdocument('http://www.example.org/books')
where $BookProperty=books:author
return <rdf:Description about=$Author>
       {
              <books:authored> $Book < / books:authored>
       }
       </ rdf:Description>
}
</ rdf:RDF>
```

Here the goal is *:-description(Description,Node,Type,Doc)* and the translation is:

```
rdf_RDF(rdf_RDFtype(Description,[Xmlns_rdf,Xmlns_books]),Node,1,"result.xml"):-
        description(Description,[NodeDescription|Node],2,"result.xml"),
        xmlns_rdf(Xmlns_rdf,Node,2,"result.xml"),
        xmlns_books(Xmlns_books,Node,2,"result.xml").
xmlns_rdf("http://www.w3.org/1999/02/22-rdf-syntax-ns#",[1],2,"result.xml").
xmlns_books("http://example.org/books#",[1],2,"result.xml").
description(descriptiontype(books_authoredtype(Book,[]),[about=Author]),Node,2,"result.xml"):-
        join(Book,Author,Node,2).
join(Book,Author,[N,1],2):-
        triple(Book,BookProperty,Author,'http://www.example.org/books',N),
        eq(BookProperty,books:author).
```

The **Query 7** does not introduce new elements in the translation.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#",
xmlns:books="http://example.org/books#">
{
for ($Book1, $BookProperty1, $Title)
              in rdfdocument('http://www.example.org/books')
return
for ($Book2, $BookProperty2, $Author)
              in rdfdocument('http://www.example.org/books')
where $Book1=$Book2
       and $BookProperty1=books:title
       and $BookProperty2=books:author
       return <rdf:Description about=$Book1>
       {
              <books:title > $Title < / books:title>
              <books:author> $Author < / books:author>
       }
       </ rdf:Description>
}
</ rdf:RDF>
```

The goal in this case is *:-description(Description,Node,Type,Doc)* and the translation is:

```
rdf_RDF(rdf_RDFtype(Description,[Xmlns_rdf,Xmlns_books]),Node,1,"result.xml"):-
        description(Description,[NodeDescription|Node],2,"result.xml"),
        xmlns_rdf(Xmlns_rdf,Node,2,"result.xml"),
        xmlns_books(Xmlns_books,Node,2,"result.xml").
xmlns_rdf("http://www.w3.org/1999/02/22-rdf-syntax-ns#",[1],2,"result.xml").
xmlns_books("http://example.org/books#",[1],2,"result.xml").
description(descriptiontype(books_titletype(Title,[]),books_authortype(Author,[]),
        [about=Book1]),Node,2,"result.xml"):-
        join(Title,Author,Book1,Node,2).
join(Title,Author,[N,1],2):-
        triple(Book1,BookProperty1,Author,'http://www.example.org/books',N),
        triple(Book2,BookProperty2,Title,'http://www.example.org/books',M),
        eq(Book1,Book2),
        eq(BookProperty1,books:title),
        eq(BookProperty2,books:author).
```

In the case of **Query 8**, a nested query is represented following the same criteria as for unnested queries, but the key point of the translation is the use of the numbering for a *correct nesting* of the output XML document. With this aim, the records for each subclass of the same class are numbered with the same number in order to be included together in the same *class* record.

```
<subclassof>
{
for ($Subject1, $Property1, $Object1) in rdfdocument('http://www.example.org/books')
where $Property1=rdf:type and $Object1=rdfs:Class
return
<class classname=$Subject1>
{
for ($Subject2, $Property2, $Object2) in rdfdocument('http://www.example.org/books')
where $Property2=rdf:type and $Object2=rdfs:Class
        and rdfs:subClassof($Subject2,$Subject1)
        return <subclass>
               {
               $Subject2
               }
               </subclass>
}
</class>
}
</subclassof>
```

In this case the goal is *:-class(Class,Node,Type,Doc)* and the translation is:

```
subclassof(subclassoftype(Class,[]),Node,1,"result.xml"):-
        class(Class,[NodeClass|Node],2,"result.xml").
class(classtype(subclasstype(Subject2,[]),[classname=Subject1]),Node,2,"result.xml"):-
        join(Subject2,Subject1,Node,2,"result.xml").
join(Subject2,Subject1,[N,1],2,"result.xml"):-
        triple(Subject1,Property1,Object1,'http://www.example.org/books',N),
        triple(Subject2,Property2,Object2,'http://www.example.org/books',M),
        eq(Property1,rdf:type),
        eq(Object1,rdfs:class),
        eq(Property2,rdf:type),
        eq(Object2,rdfs:class),
        rdfs_subClassof( Subject2, Subject1,'http://www.example.org/books').
```

5 Conclusions and Future Work

In this paper we have studied an extension of *XQuery* for the handling of RDF documents. Such extension combines RDF and XML documents as input/output documents. By means of built-in predicates *XQuery* can be equipped with inference mechanism for RDF properties. We have also studied how to implement such language in logic programming. The proposed translation can be generalized and achieved in an automatic way. We are now developing a formal translation of the extended *XQuery* into logic programming in order to be implemented. We would like to develop a prototype of our language using the SWI-Prolog platform and the RDF library. We can take advantage from the RDF storing, retrieval and inferring of SWI-Prolog [30] for the implementation of the proposed extension of *XQuery* into *Prolog*. In addition, we would like to study how to use logic programming for enriching modeling and inference based on more complex ontology languages like OWL and how to integrate it with our proposal.

References

[1] J. M. Almendros-Jiménez, A. Becerra-Terón, and Francisco J. Enciso-Baños. Magic sets for the XPath language. *Journal of Universal Computer Science*, 12(11):1651–1678, 2006.

[2] J. M. Almendros-Jiménez, A. Becerra-Terón, and Francisco J. Enciso-Baños. Integrating XQuery and Logic Programming. In *Proceedings of the Workshop on Logic Programming*, pages 136–147, Germany, 2007. Technical Report 434, University of Würzburg.

[3] J. M. Almendros-Jiménez, A. Becerra-Terón, and Francisco J. Enciso-Baños. Querying XML documents in logic programming. *Journal of Theory and Practice of Logic Programming, available at* `http://www.ual.es/~jalmen`, *in press*, 2008.

[4] James Bailey, François Bry, Tim Furche, and Sebastian Schaffert. Web and Semantic Web Query Languages: A Survey. In *Proc. of Reasoning Web, First International Summer School*, pages 35–133, Heidelberg, Germany, 2005. LNCS 3564.

[5] Tim Berners-Lee. N3QL-RDF Data Query Language. Technical report, Online only, 2004.

[6] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web – A new form of Web content that is meaningful to computers will unleash a revolution of new possibilities. *Scientic American*, May:36 pages, 2001.

[7] H. Boley. Relationships Between Logic Programming and RDF. In *Proc. of Advances in Artificial Intelligence*, pages 201–218, Heidelberg, Germany, 2000. LNCS 2112.

[8] D. Chamberlin, Denise Draper, Mary Fernández, Michael Kay, Jonathan Robie, Michael Rys, Jerome Simeon, Jim Tivy, and Philip Wadler. *XQuery from the Experts*. Addison Wesley, Boston, USA, 2004.

[9] I. Davis. RDF Template Language 1.0. Technical report, Online only, September 2003.

[10] Tim Furche, François Bry, and Oliver Bolzer. Marriages of Convenience: Triples and Graphs, RDF and XML in Web Querying. In *Proceedings of Third Workshop on Principles and Practice of Semantic Web Reasoning, Dagstuhl, Germany (11th–16th September 2005)*, volume 3703 of *LNCS*, pages 72–84. REWERSE, 2005.

[11] Peter Haase, Jeen Broekstra, Andreas Eberhart, and Raphael Volz. A Comparison of RDF query languages. In Sheila A. McIlraith, Dimitris Plexousakis, and Frank van Harmelen, editors, *Proceedings of the Third International Semantic Web Conference, Hiroshima, Japan, 2004*, volume 3298, pages 502–517. Springer Berlin / Heidelberg, November 2004.

[12] Gregory Karvounarakis, Sofia Alexaki, Vassilis Christophides, Dimitris Plexousakis, and Michel Scholl. RQL: a declarative query language for RDF. In *WWW '02: Proceedings of the 11th international conference on World Wide Web*, pages 592–603, New York, NY, USA, 2002. ACM Press.

[13] H. Katz. XsRQL: an XQuery-style Query Language for RDF. Technical report, Online only, 2004.

[14] Patrick Lehti and Peter Fankhauser. XML Data Integration with OWL: Experiences and Challenges. In *Procs. of SAINT'04*, page 160, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[15] Libby Miller, Andy Seaborne, and Alberto Reggiori. Three Implementations of SquishQL, a Simple RDF Query Language. In *ISWC '02: Proceedings of the First International Semantic Web Conference on The Semantic Web*, pages 423–435, London, UK, 2002. Springer-Verlag.

[16] Cristian Pérez de Laborda and Stefan Conrad. Bringing Relational Data into the Semantic Web using SPARQL and Relational OWL. In *Procs. of ICDEW'06*, page 55, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[17] Peter F. Patel-Schneider and Simeon Simeon. The Yin/Yang Web: A Unified Model for XML Syntax and RDF Semantics. *IEEE Transactions on Knowledge and Data Engineering*, 15(4):797–812, 2003.

[18] Jonathan Robie, Lars Marius Garshol, Steve Newcomb, Michel Biezunski, Matthew Fuchs, Libby Miller, Dan Brickley, Vassilis Christophides, and Gregorius Karvounarakis. The Syntactic Web: Syntax and Semantics on the Web. *Markup Languages: Theory & Practice*, 4(3):411–440, 2002.

[19] S. Schaffert and F. Bry. A Gentle Introduction to Xcerpt, a Rule-based Query and Transformation Language for XML. In *Proc. of International Workshop on Rule Markup Languages for Business Rules on the Semantic Web*, page 22 pages, Aachen, Germany, 2002. CEUR Workshop Proceedings 60.

[20] Michael Sintek and Stefan Decker. TRIPLE - A Query, Inference, and Transformation Language for the Semantic Web. In *ISWC '02: Proceedings of the First International Semantic Web Conference on The Semantic Web*, pages 364–378, London, UK, 2002. Springer-Verlag.

[21] Adam Souzis. RxPath: a mapping of RDF to the XPath Data Model. In *Extreme Markup Languages*, 2006.

[22] Igor Tatarinov, Stratis D. Viglas, Kevin Beyer, Jayavel Shanmugasundaram, Eugene Shekita, and Chun Zhang. Storing and Querying Ordered XML using a Relational Database System. In *Proc. of the ACM SIGMOD Conference*, pages 204–215, New York, USA, 2002. ACM Press.

[23] W3C. OWL Ontology Web Language. Technical report, www.w3.org, 2004.

[24] W3C. RDF Vocabulary Description Language 1.0: RDF Schema. Technical report, www.w3.org, 2004.

[25] W3C. Resource Description Framework (RDF). Technical report, www.w3.org, 2004.

[26] W3C. XML Path Language (XPath) 2.0. Technical report, www.w3.org, 2007.

[27] W3C. XML Query Working Group and XSL Working Group, XQuery 1.0: An XML Query Language. Technical report, www.w3.org, 2007.

[28] Norman Walsh. RDF Twig: Accessing RDF Graphs in XSLT. In *Proceedings of Extreme Markup Languages*, 2003.

[29] Raphael Wolz. *Web Ontology Reasoning with Logic Databases*. PhD thesis, Universität Fridericiana zu Karlsruhe, 2004.

[30] Jan Wielemaker, Guus Schreiber, and Bob J. Wielinga. Prolog-Based Infrastructure for RDF: Scalability and Performance. In Dieter Fensel, Katia P. Sycara, and John Mylopoulos, editors, *International Semantic Web Conference*, volume 2870 of *Lecture Notes in Computer Science*, pages 644–658. Springer, 2003.