Querying XML Data with SPARQL*

Nikos Bikakis, Nektarios Gioldasis, Chrisa Tsinaraki,
Stavros Christodoulakis

Technical University of Crete, Department of Electronic and Computer Engineering
Laboratory of Distributed Multimedia Information Systems & Applications (TUC/ MUSIC)
University Campus, 73100, Kounoupidiana Chania, Greece
{nbikakis, nektarios, chrisa, stavros}@ced.tuc.gr

Abstract. SPARQL is today the standard access language for Semantic Web data. In the recent years XML databases have also acquired industrial importance due to the widespread applicability of XML in the Web. In this paper we present a framework that bridges the heterogeneity gap and creates an interoperable environment where SPARQL queries are used to access XML databases. Our approach assumes that fairly generic mappings between ontology constructs and XML Schema constructs have been automatically derived or manually specified. The mappings are used to automatically translate SPARQL queries to semantically equivalent XQuery queries which are used to access the XML databases. We present the algorithms and the implementation of *SPARQL2XQuery* framework, which is used for answering SPARQL queries over XML databases.

Keywords: Semantic Web, XML Data, Information Integration, Interoperability, Query Translation, SPARQL, XQuery, SPARQL to XQuery translation/transformation, SPARQL2XQuery.

1 Introduction

The Semantic Web has to coexist and interoperate with other software environments and in particular with legacy databases. The *Extensible Markup Language (XML)*, its derivatives (*XPath*, *XSLT*, etc.), and the *XML Schema* have been extensively used to describe the syntax and structure of complex documents. In addition, XML Schema has been extensively used to describe the standards in many business, service, and multimedia application environments. As a result, a large volume of data is stored and managed today directly in the XML format in order to avoid inefficient access and conversion of data, as well as avoiding involving the application users with more than one data models. The database management systems offer today an environment supporting the XML data model and the XQuery access language for managing XML data. In the Web application environment the XML Schema acts also as a wrapper to relational content that may coexist in the databases.

Our working scenario assumes that users and applications of the Semantic Web environment ask for content from underlying XML databases using SPARQL. The

* An extended version of this paper is available at [14].

SPARQL queries are translated into semantically equivalent XQuery queries which are (exclusively) used to access and manipulate the data from the XML databases in order to return the requested results to the user or the application. The results are returned in RDF (N3 or XML/RDF) or XML [1] format. To answer the SPARQL queries on top of the XML databases, a mapping at the schema level is required. We support a set of language level correspondences (rules) for mappings between RDFS/OWL and XML Schema. Based on these mappings our framework is able to translate SPARQL queries into semantically equivalent XQuery expressions as well as to convert XML Data in the RDF format. Our approach provides an important component of any Semantic Web middleware, which enables transparent access to existing XML databases.

The framework has been smoothly integrated with the *XS2OWL* framework [9], thus achieving not only the automatic generation of mappings between XML Schemas and OWL ontologies, but also the transformation of XML documents in RDF format.

Various attempts have been made in the literature to address the issue of accessing XML data from within Semantic Web Environments [2, 4, 5, 6, 7, 8, 9, 10, 11, 12]. An extended overview of related work can be found at [13].

The rest of the paper is organized as follows: The mappings used for the translation as well as their encoding are described in Section 2. Section 3 provides an overview of the query translation process. The paper concludes in section 4.

2 Mapping OWL to XML Schema

The framework described here allows XML encoded data to be accessed from Semantic Web applications that are aware of some ontology encoded in OWL. To do that, appropriate mappings between the OWL ontology (*O*) and the XML Schema (*XS*) should exist. These mappings may be produced either automatically, based on our previous work in the *XS2OWL* framework [9], or manually through some mapping process carried out by a domain expert. However, the definition of mappings between OWL ontologies and XML Schemas is not the subject of this paper. Thus, we do not focus on the semantic correctness of the defined mappings. We neither consider what the mapping process is, nor how these mappings have been produced

Such a mapping process has to be guided from language level correspondences. That is, the valid correspondences between the OWL and XML Schema language constructs have to be defined in advance. The language level correspondences that have been adopted in this paper are well-accepted in a wide range of data integration approaches [2, 4, 9, 10, 11]. In particular, we support mappings that obey the following language level correspondence rules: A class of *O* corresponds to a Complex Type of *XS*, a DataType Property of *O* corresponds to a Simple Element or Attribute of *XS*, and an Object Property of *O* corresponds to a Complex Element of *XS*.

Then, at the schema level, mappings between concrete domain conceptualizations have to be defined (e.g. the *employee* class is mapped to the *worker* complex type) following the correspondences established at the language level.

At the schema level mappings a mapping relationship between *O* and an *XS* is a binary association representing a semantic association among them. It is possible that

for a single ontology construct more than one mapping relationships are defined. That is, a single source ontology construct can be mapped to more than one target XML Schema elements (1:n mapping) and vice versa, while more complex mapping relationships can be supported.

The mappings considered in our work are based on the *Consistent Mappings Hypothesis*, which states that for each mapped property *Pr* of *O*:

a. The domain classes of *Pr* have been mapped to complex types in *XS* that contain the elements or attributes that *Pr* has been mapped to.
b. If *Pr* is an object property, the range classes of *Pr* have been mapped to complex types in *XS*, which are used as types for the elements that *Pr* has been mapped to.

2.1 Encoding of the Schema Level Mappings

Since we want to translate SPARQL queries into semantically equivalent XQuery expressions that can be evaluated over XML data following a given (mapped) schema, we are interested in addressing XML data representations. Thus, based on schema level mappings for each mapped ontology class or property, we store a set of XPath expressions ("*XPath set*" for the rest of this paper) that address all the corresponding instances (XML nodes) in the XML data level. In particular, based on the schema level mappings, we construct:

- A **Class XPath Set** X_C for each mapped class *C*, containing all the possible XPaths of the complex types to which the class *C* has been mapped to.
- A **Property XPath Set** X_{Pr} for each mapped property *Pr*, containing all the possible XPaths of the elements or/and attributes to which *Pr* has been mapped.

For ontology properties, we are also interested in identifying the property domains and ranges. Thus, for each property we define the X_{PrD} and X_{PrR} sets, where:

- The **Property Domains XPath Set** X_{PrD} for a property *Pr* represents the set of the XPaths of the property domain classes.
- The **Property Ranges XPath Set** X_{PrR} for a property *Pr* represents the set of the XPaths of the property ranges.

Example 1. Encoding of Mappings

Fig. 1 shows the mappings between an OWL Ontology and an XML Schema.

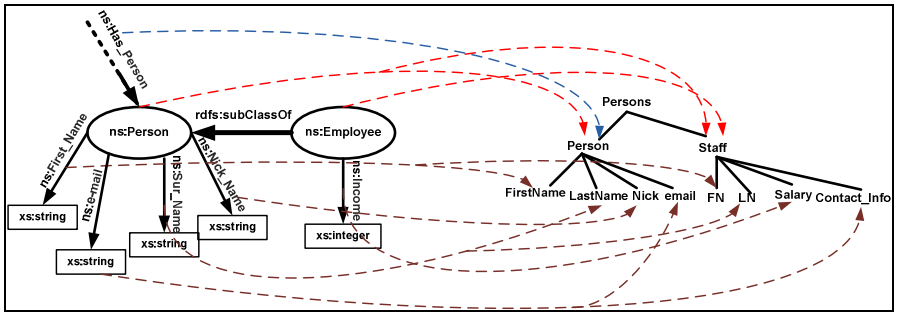

Fig. 1. Mappings Between OWL & XML

To better explain the defined mappings, we show in Fig. 1 the structure of the XML documents that follow this schema. The encoding of these mappings in our framework is shown in Fig.2.

Classes:	**DataType Properties:**
$\mathbf{X}_{\mathbf{ns:Person}}$={/Persons/Person, /Persons/Staff}	$\mathbf{X}_{\mathbf{ns:First\_Name}}$={/Persons/Person/FirstName, /Persons/Staff/FN}
$\mathbf{X}_{\mathbf{ns:Employee}}$={/Persons/Staff}	$\mathbf{X}_{\mathbf{ns:Sur\_Name}}$={/Persons/Person/LastName, /Persons/Staff/LN}
	$\mathbf{X}_{\mathbf{ns:Nick\_Name}}$={/Persons/Person/Nick }
Object Properties:	$\mathbf{X}_{\mathbf{ns:e\text{-}mail}}$={/Persons/Person/email, /Persons/Staff/Contact_Info}
$\mathbf{X}_{\mathbf{ns:Has\_Person}}$={/Persons/Person }	$\mathbf{X}_{\mathbf{ns:Income}}$={/Persons/Staff/Salary}

Fig. 2. Mappings Encoding

XPath Set Operators. For XPath Sets, the following operators are defined in order to formally explain the query translation methodology in the next sections:

- The unary ***Parent Operator*** ^{P}, which, when applied to a set of XPaths *X* (i.e. $(X)^{P}$), returns the set of the distinct parent XPaths (i.e. the same XPaths without the leaf node). When applied to the root node, the operator returns the same node.

Example 2. Let X={ /a , /a/b , /c/d , /e/f/g , /b/@f } then $(X)^{P}$={ /a , /a , /c , /e/f , /b }.

- The binary ***Right Child Operator*** ®, which, when applied to two XPath sets *X* and *Y* (i.e. *X*®*Y*), returns the members (XPaths) of the right set *X*, the parent XPaths of which are contained in the left set *Y*.

Example 3. Let X={ /a , /c/b } and Y={ /a/d , /a/c , /c/b/p , c/a/g } then
X ®Y = { /a/d , /a/c , /c/b/p } .

- The binary ***Append Operator*** /, which is applied on an XPath set *X* and a set of node names *N* (i.e. *X / N*), resulting in a new set of XPaths *Y* by appending each member of *N* to each member of *X*.

Example 4. Let X={/a, /a/b} and N={c, d} then Y = X / N = {/a/c, /a/d, /a/b/c, a/b/d }.

XPath Set Relations. We describe here a relation among XPath sets that holds because of the *Consistent Mapping Hypothesis* described above. We will use this relation later on in the query translation process, and in particular in the variable bindings algorithm (subsection 3.1):

Domain-Range Property Relation: $\forall \text{ Property } \Pr \Rightarrow X_{\Pr R} = X_{\Pr}$ and $X_{\Pr D} = \left(X_{\Pr}\right)^{P} = \left(X_{\Pr R}\right)^{P}$

The *Domain-Range Property Relation* can be easily understood taking into account the hierarchical structure of XML data as well as the *Consistent Mappings Hypothesis*. It describes that for a single property *Pr*:

- the XPath set of its ranges is equal to its own XPath set (i.e. the instances of its ranges are the XML nodes of the elements that this property has been mapped to).
- the XPath set of its domain classes is equal to the set containing its parent XPaths (i.e. the XPaths of the *CTs(Complex Types)* that contain the elements that this property has been mapped to).

3 Overview of the Query Translation Process

In this section we present in brief the entire translation process using a UML activity diagram. Fig. 3 shows the entire process which starts taking as input the given SPARQL query and the defined mappings between the ontology and the XML Sche-

ma (encoded as described in the previous sections). The query translation process comprises of the activities outlined in the following paragraphs.

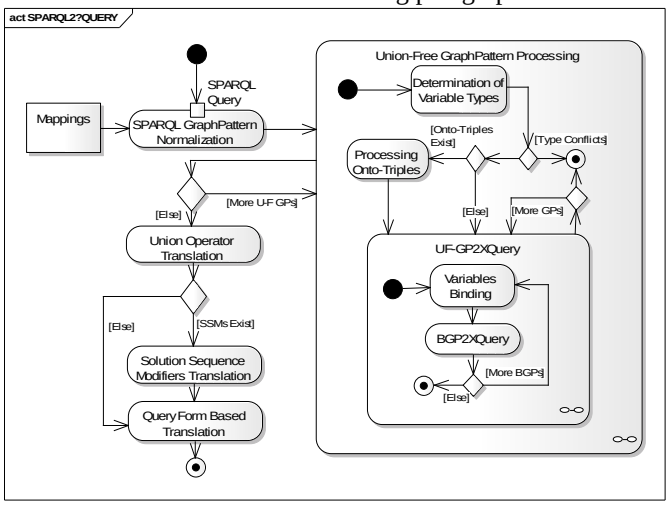

Fig. 3. Overview of the SPARQL Translation Process

SPARQL Graph Pattern Normalization. The *SPARQL Graph Pattern Normalization* activity re-writes the Graph-Pattern (*GP*) of the SPARQL query in an equivalent normal form based on equivalence rules. The SPARQL *GP* normalization is based on the *GP* expression equivalences proved in [3] and re-writing techniques. In particular, each *GP* can be transformed in a sequence *P1 UNION P2 UNION P3 UNION…UNION Pn*, where *Pi* ($1 \leq i \leq n$) is a Union-Free *GP* (i.e. *GPs* that do not contain Union operators). This makes the *GP* translation process simpler and more efficient.

Union-Free Graph Pattern (UF-GP) Processing. The *UF-GP* processing translates the constituent *UF-GPs* into semantically equivalent XQuery expressions. The *UF-GP* Processing activity is a composite one, with various sub-activities. This is actually the step that most of the "real work" is done since at this step most of the translation process takes place. The *UF-GP processing* activity is decomposed in the following sub-activities:

– Determination of Variable Types. For every *UF-GP*, this activity initially identifies the types of the variables used in order to detect any conflict arising from the user's syntax of the input as well as to identify the form of the results for each variable. We define the following variable types: The *Class Instance Variable Type* (*CIVT*), The *Literal Variable Type* (*LVT*), The *Unknown Variable Type* (*UVT*), The *Data Type Predicate Variable Type* (*DTPVT*), The *Object Predicate Variable Type* (*OPVT*), The *Unknown Predicate Variable Type* (*UPVT*).

We also define the following sets: The Data *Type Properties Set* (*DTPS*), which contains all the data type properties of the ontology. The *Object Properties Set* (*OPS*), which contains all the object properties of the ontology. The *Variables Set* (*V*), which contains all the variables that are used in the *UF-GP*. The *Literals Set* (*L*), which contains all the literals referenced in the *UF-GP*.

The determination of the variable types is based on a set of rules applied iteratively for each triple in the given *UF-GP*. Below we present a subset of these rules, which are used to determine the type (T_X) of a variable *X*:

Let *S P O* be a triple pattern.

1. If $P \in OPS$ and $O \in V \Rightarrow T_O = CIVT$. If predicate is an object property and object is a variable, then the type of the object variable is *CIVT*.
2. If $O \in L$ and $P \in V \Rightarrow T_P = DTPVT$. If the object is a literal value, then the type of the predicate variable is *DTPVT*.

– Processing Onto-Triples. *Onto-Triples* actually refer to the ontology structure and/or semantics. The main objective of this activity is to process *Onto-Triples* against the ontology (using SPARQL) and based on this analysis to bind (i.e. assigning the relevant XPaths to variables) the correct XPaths to variables contained in the *Onto-Triples*. These bindings are going to be used in the next steps as input to the *Variable Bindings* activity.

– UF-GP2XQuery. This activity translates the *UF-GP* into semantically equivalent XQuery expressions. The concept of a *GP*, and thus the concept of *UF-GF*, is defined recursively. The *BGP2XQuery* algorithm translates the basic components of a *GP* (i.e. Basic Graph Patterns - *BGPs* which are sequences of triple patterns and filters) into semantically equivalent XQuery expressions (see subsection 3.2). To do that a variables binding (see subsection 3.1) step is needed. Finally, *BGPs* in the context of a *GP* have to be properly associated. That is, to apply the SPARQL operators among them using XQuery expressions and functions. These operators are: *OPT*, *AND*, and *FILTER* and are implemented using standard XQuery expressions without any ad hoc processing.

Union Operator Translation. This activity translates the *UNION* operator that appears among *UF-GPs* in a *GP*, by using the *Let* and *Return* XQuery clauses in order to return the union of the solution sequence produced by the *UF-GPs* to which the Union operator applies.

Solution Sequence Modifiers Translation. This activity translates the SPARQL solution sequence modifiers using XQuery clauses (*Order By*, *For*, *Let*, etc.) and XQuery built-in functions (you can see the example in subsection 3.3.). The modifiers supported by SPARQL are *Distinct*, *Order By*, *Reduced*, *Limit*, and *Offset*.

Query Forms Based Translation. SPARQL has four forms of queries (*Select*, *Ask*, *Construct* and *Describe*). According to the query form, the structure of the final result is different. The query translation is heavily dependent on the query form. In particular, after the translation of any solution modifier is done, the generated XQuery is enhanced with appropriate expressions in order to achieve the desired structure of the results (e.g. to construct an RDF graph, or a result set) according to query form.

3.1 Variable Bindings

This section describes the variable bindings activity. In the translation process the term *"variable bindings"* is used to describe the assignment of the correct XPaths to the variables referenced in a given *Basic Graph Pattern* (*BGP*), thus enabling the translation of *BGP* to XQuery expressions. In this activity, *Onto-Triples* are not taken into account since their processing has taken place in the previous step.

Definition 1 : A triple pattern has the form $(s,p,o) \in (I \cup B \cup V) \times (I \cup V \cup B) \times (I \cup B \cup L \cup V)$, where I is a set of IRIs, B is a set of Blank Nodes, V is a set of Variables, and L the set of RDF Literals. In our approach, however, the individuals in the source ontology are not considered at all (either they do not exist, or they are not used in semantic queries).

Definition 2 : A variable contained in a Union Free Graph Pattern is called a *Shared Variable* when it is referenced in more than one triple patterns of the same Union-Free Graph Pattern regardless its position in those triple patterns.

Variable Bindings Algorithm. When describing data with the RDF triples (*s,p,o*), subjects represent class individuals (RDF nodes), predicates represent properties (RDF arcs), and objects represent class individuals or data type values (RDF nodes). Based on that, and the *domain-range property* relation of Xpaths sets relations section we have: **a)** $X_s = X_{pD} = (X_{pR})^P = (X_p)^P$ **b)** $X_p = X_{pR}$ and **c)** $X_o = X_{pR}$.
Thus it holds that: $X_s = X_{pD} = (X_{pR})^P = (X_p)^P = (X_o)^P \Rightarrow X_s = (X_p)^P = (X_o)^P$ ***(Subject-Predicate-Object Relation)***

This relation holds for every single triple pattern. Thus, the variable bindings algorithm uses this relation in order to find the correct bindings for the entire set of triple patterns starting from the bindings of any single triple pattern part (subject, predicate, or object).

In case of shared variables, the algorithm tries to find the maximum set of bindings (using the operators for XPath sets) that satisfy this relation for the entire set of triple patterns (e.g. the entire *BGP*). Once this relation holds for the entire *BGP* we have as a result that all the instances (in XML) that satisfy the *BGP* have been addressed.

The variable bindings algorithm in case of shared variables of *LVT* type it doesn't determine the XPaths for this kind of variable, since literal equality is independent of the XPaths expressions. Thus, the bindings for variables of this type cannot be defined at this step (mark as *"Not Definable"* at variable bindings rules). Instead, they will be handled by the *BGP2XQuery* (subsection 3.2) algorithm (using the mappings and the determined variables bindings).

The algorithm takes as input a *BGP* as well as a set of initial bindings and the types of variables as these are determined in the "*Determination of Variable Type*" activity. These initial bindings are the ones produced by the *Onto-Triple* processing activity and initialize the bindings of the algorithm. Then, the algorithm performs an iterative process where it determines, at each step, the bindings of the entire *BGP* (triple by triple). The determination of the bindings is based on the rules described below. This iterative process continues until the bindings for all the variables found in the successive iterations are equal. This means that no further modifications in the variable bindings are to be made and that the current bindings are the final ones.

Variable Bindings Rules. Based on the possible combinations of *S, P* and *O*, there are four different types of triple patterns (the ontology instance are not yet supported by our framework): **Type 1 :** $S \in V, P \in I, O \in L$. **Type 2 :** $S, O \in V, P \in I$. **Type 3 :** $S, P \in V, O \in L$. **Type 4 :** $S, P, O \in V$.

According to the triple pattern type, we have defined a set of rules for the variable bindings. In this section we present a sub-set of these rules due to space limitations.

In what follows the symbol $'$ in XPath sets denotes the new bindings assigned to the set at each iteration, while the symbol $\leftarrow$ denotes the assignment of a new value to the set. All the XPath sets are considered to be initially set to *null*. In that case, the intersection operation is not affected by the *null* set. E.g. X={ null } and Y= {/a/b , d/e} then X $\cap$ Y ={ /a/b , d/e }. The notation *"Not Definable"* is used for variables of type *LVT* as explained above. Consider the triple *S P O* :

- If the triple is of Type 1 $\Rightarrow X_S' \leftarrow X_{PD} \cap X_S$
- If the triple is of Type 2 $\Rightarrow X_S' \leftarrow X_{PD} \cap X_S \cap (X_O)^P$
  - If $P \in OPS \Rightarrow X_O' \leftarrow X_S'$ ® X_O
  - If $P \in DTPS \Rightarrow X_O'$ *Non Definable* (as explained in previously)
- If the triple is of Type 3 $\Rightarrow X_S' \leftarrow X_{PD} \cap X_S$ and $X_P' \leftarrow X_S'$ ® X_P
- If the triple is of Type 4 $\Rightarrow X_S' \leftarrow X_{PD} \cap X_S \cap (X_O)^P$ and $X_P' \leftarrow X_S'$ ® X_P
  - If $T_O = CIVT$ or $T_O = UVT \Rightarrow X_O' \leftarrow X_P' \cap X_O$
  - If $T_O = LVT \Rightarrow X_O'$ *Non Definable* (as explained previously)

XPath Set Relations for Triple-Patterns. Among XPath sets of triple patterns there are important relations that can be exploited in the development of the XQuery expressions in order to correctly associate data that have been bound to different variables of triple patterns. The most important relation among XPath sets of triple patterns is that of extension:

Extension Relation: An XPath set *A* is said to be an extension of an XPath set *B* if all XPaths in *A* are descendants of the XPaths of *B*.

As an example of this relation, consider the XPath A' produced when applying the append (/) operator to an original XPath set *A* with a set of nodes.

The extension relation holds for the results of the variable bindings algorithm (*Subject-Predicate-Object Relation*) and implies that the XPaths bound to subjects are parents of the XPaths bound to predicates and objects of triple patterns.

3.2 Translating BGPs to XQuery

In this section we describe the translation of *BGPs* to semantically equivalent XQuery expressions. The algorithm manipulates a sequence of triple patterns and filters (i.e. a *BGP*) and translates them into semantically equivalent XQuery expressions, thus allowing the evaluation of a *BGP* on a set of XML data.

Definition 3 : *Return Variables* (*RV*) are those variables for which the given SPARQL Query would return some information. The set of all *Return Variables* of a SPARQL query constitutes the set $RV \subseteq V$.

The BGP2XQuery Algorithm. We briefly present here the *BGP2XQuery* algorithm for translating *BGPs* into semantically equivalent XQuery expressions. The algorithm takes as input the mappings between the ontology and the XML schema, the *BGP*, the determined variable types, as well as the variable bindings. The algorithm is not executed triple-by-triple for a complete *BGP*. Instead, it processes subjects, predicates, and objects of all the triples separately. For each variable included in the *BGP*, the *BGP2XQuery* it creates a *For* or *Let* XQuery clause using the variable

bindings, the input mappings, and the *Extension Relation* for triple-patterns (see subsection.3.1), in order to bound XML data into XQuery variables. The choice between the *For* and the *Let* XQuery clauses is based on specific rules so as to create a solution sequence based on the SPARQL semantics. Moreover, in order to associate bindings from different variables into concrete solutions, the algorithm uses the *Extension Relation*. For literals included in the *BGP*, the algorithm is using XPath predicates in order to translate them. Due to the complexity that a SPARQL filter may have, the algorithm translates all the filters into XQuery where clauses, although some "simple" of them (e.g. condition on literals) could be translated using XPath predicates. Moreover, SPARQL operators (Built-in functions) included in filter expressions are translated using built-in XQuery functions and operators. However, for some "special" SPARQL operators (like *sameTerm*, *lang*, etc.) we have developed native XQuery functions that simulate them.

Finally, the algorithm creates an XQuery *Return* clause that includes the Return Variables (*RV*) that was used in the *BGP*.

There are some cases of share variables which need special treatment by the algorithm in order to apply the required joins in XQuery expressions. The way that the algorithm manipulates these cases depends on which parts (*subject-predicate-object*) of the triples patterns these shared variables refer to.

3.3 Example

We demonstrate in this example the use of the described framework in order to allow a SPARQL query to be evaluated in XML Data (based on Example 1). Fig. 4 shows how a given SPARQL query is translated by our framework into a semantically equivalent XQuery.

Consider the query: "*Return the Persons, their last name(s) and their nick name(s), whose first name is "John", whose last name begins with "A", and they have an e-mail address. The (existence of) nick name is optional. The query will return at most 30 solutions (LIMIT 30) ordered by last name value at descending order and skipping the first 5 solutions (OFFSET 5)*".

SPARQL Query:

```
PREFIX ns:<http://example.com/ns#>

SELECT  ?x ?lname ?nick
WHERE{ { ?x   ns:First_Name  "John".          } BGP_1
         ?x   ns:Sur_Name    ?lname .         }
         ?x   ns:email       ?email.          }
         FILTER regex( ?lname, "^A")  }       }
       OPTIONAL{ ?x  ns:Nick_Name  ?nick. }   } BGP_2
       }
ORDER BY DESC (?lname)
 LIMIT 30  OFFSET 5
```

Translated XQuery Query :

```
declare namespace func = "http://www.music.tuc.gr/funcs";
let $doc := collection("http://www.music.tuc.gr/....")
let $Modified_Results :=(
   let $Results :=(
      let $BGP_1:=(
         for $x in $doc/Persons/Person[./FirstName= "John" ] union
                  $doc/Persons/Staff[./FN= "John" ]
         for $lname in $x/LastName union $x/LN
         let $email := $x/email  union $x/Contact_Info
         where( exists($email) and matches($lname, "^A" ) )
         return(<Result> <x>{func:CIVT($x)}</x>,
                              <lname>{ string($lname)}</lname></Result>)
      )
      let $BGP_2:=(
         for $x in $doc/Persons/Person
         for $nick in $x/Nick
         return(<Result> <x>{func:CIVT($x)}</x>,
                              <nick>{ string($nick)}</nick></Result>)
      )
      return ( func:OPTIONAL($BGP_1, $BGP_2) )
   )
   return (let $Ordered_Results:=(
                              for $iter in $Results
                              order by $iter/lname descending empty least
                              return($iter) )
            return ($Ordered_Results[position( )>5 and position( )<=35]))
)
return (<Results>{$Modified_Results }</Results>)
```

Fig. 4. SPARQL Query Translation Example

4 Conclusions

We have presented a framework and its software implementation that allows the evaluation of SPARQL queries over XML data which are stored in XML databases and accessed with the XQuery language. The framework assumes that a set of mappings between the OWL ontology and the XML Schema exists which obey to certain well accepted language correspondences.

The *SPARQL2XQuery* framework has been implemented as a software service which can be configured with appropriate mappings (between some ontology and XML Schema) and translates input SPARQL queries into semantically equivalent XQuery queries that are answered over the XML Database.

5 References

1. Beckett D. (eds), "SPARQL Query Results XML Format". W3C Recommendation, 15 January 2008, (*http://www.w3.org/TR/rdf-sparql-XMLres/*).
2. Bohring H., Auer S.: "Mapping XML to OWL Ontologies". Leipziger Informatik-Tage 2005: 147-156
3. J. Perez, M. Arenas, C. Gutierrez. Semantics and Complexity of SPARQL. 5th International Semantic Web Conference (ISWC-06), November 2006.
4. Rodrigues T., Rosa P, Cardoso J., "Mapping XML to Exiting OWL ontologies", International Conference WWW/Internet 2006, Murcia, Spain, 5-8 October 2006.
5. Joel Farrell and Holger Lausen. Semantic Annotations for WSDL and XML Schema. W3C Recommendation, W3C, August 2007. Available at http://www.w3.org/TR/sawsdl/
6. Sven Groppe, Jinghua Groppe, Volker Linnemann, Dirk Kukulenz, Nils Hoeller, Christoph Reinke: Embedding SPARQL into XQuery/XSLT. SAC 2008: 2271-2278
7. Waseem Akhtar, Jacek Kopecký et.al : XSPARQL: Traveling between the XML and RDF Worlds - and Avoiding the XSLT Pilgrimage. ESWC 2008:432-447
8. Matthias Droop, Markus Flarer et.al : "Embedding XPATH Queries into SPARQL Queries" In Proc. of the 10th International Conference on Enterprise Information Systems
9. Tsinaraki C., Christodoulakis S., "Interoperability of XML Schema Applications with OWL Domain Knowledge and Semantic Web Tools". In Proc. of the ODBASE 2007.
10. Cruz I.R., Huiyong Xiao, Feihong Hsu: "An Ontology-based Framework for XML Semantic Integration", Database Engineering and Applications Symposium, 2004.
11. V.Christophides, G. Karvounarakis et.al : "The ICS-FORTH SWIM: A Powerful Semantic Web Integration Middleware". In Proc. of the SWDB 2003, pages 381-393.
12. Bernd Amann, Catriel Beeri, Irini Fundulaki, Michel Scholl: Querying XML Sources Using an Ontology-Based Mediator. CoopIS/DOA/ODBASE 2002: 429-448
13. Bikakis N., Gioldasis N., Tsinaraki C., Christodoulakis S.: "Semantic Based Access over XML Data" In Proc. of 2nd World Summit on Knowledge Society 2009 (WSKS2009).
14. Bikakis N., Gioldasis N., Tsinaraki C., Christodoulakis S.: "The SPARQL2XQuery Framework" Technical Report *http://www.music.tuc.gr/reports/SPARQL2XQUERY.PDF*