Towards Querying Implicit Knowledge in XML Documents

Diego Mury Gomes de Lima[1], Carla Delgado[1], Leonardo Murta[2], Vanessa Braganholo[2]

[1] PPGI/Federal University of Rio de Janeiro, Brazil
dmlima@ufrj.br, carla@dcc.ufrj.br
[2] Fluminense Federal University, Brazil
{leomurta, vanesssa}@ic.uff.br

Abstract. The expressive growth in the volume of data stored as XML stimulates the improvement of methods for obtaining novel ways to explore this information. Methods for providing elaborated answers to increasingly more complex queries were recently investigated, but existing approaches mainly analyze information that is explicitly defined in the XML document. This work presents an approach that is capable of processing implicit knowledge through inference, guided by user defined rules. This can cause a significant increase in the query possibilities, allowing easy access to new information, serendipitous results and providing for more elaborate queries.

Categories and Subject Descriptors: H. Information Systems [**H.2 Database Management**]: Systems—*Rule-based databases*

Keywords: inference, prolog, query, XML

1. INTRODUCTION

XML has quickly become a universal standard for information exchange over the Web. The rise of large data collections in this format puts in evidence the need of improving query methods, so that large portions of data can be efficiently managed [Kotsakis 2002].

Most of the XML query languages currently available, such as XPath [Clark and Derose 1999] and XQuery [Boag et al. 2010], gather the query answers by navigating from the root of the document to an internal element of interest, and applying filters to obtain the desired result. The query is usually processed over the explicit data, ignoring any implicit information that may be obtained from it. This implicit information, however, is crucial in many applications. Consider, for instance, provenance data collected from workflow executions [Freire et al. 2008; Mattoso et al. 2010]. In this scenario, there is a lot of implicit information that cannot be easily queried, such as "what were the activities that contributed to produce a given workflow output?".

Stimulated by the expressive power of XML and by the large volume of information currently available in this format, the approach we propose in this work suggests an improvement in the XML query mechanisms. This is obtained through the use of inference machines in order to increase query possibilities, allowing for more elaborate results. As an example, an XML document that defines that John is the parent of Peter, and that Peter is the parent of Paul, does not have enough information to answer queries about grandparenthood. However, once an expert inserts a rule that specifies that grandparent is the parent's parent, the system should be able to answer that John is Paul's grandparent.

To make inference possible, we need to translate XML data to some language that provides such capability. RuleML [Boley 2001], Datalog [Gallaire and Minker 1978], RIF [Kifer 2008] and Prolog

Authors would like to thank CNPq and FAPERJ for partially supporting this research work.

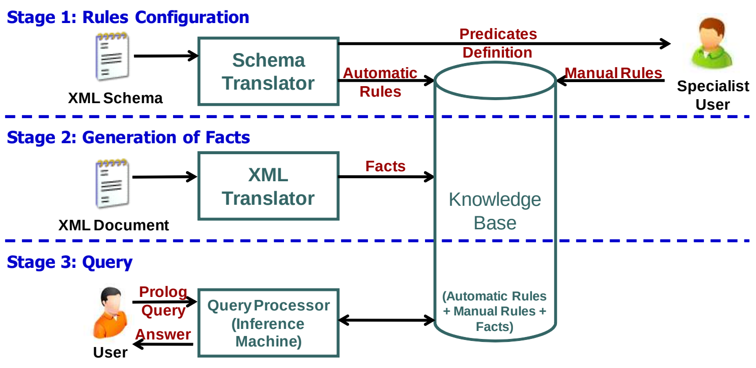

Fig. 1. Stages of the proposed approach

[Lloyd 1984] allow inference and are alternatives to solve this problem. Our choice was based on the features of these languages. Datalog, for instance, does not accept complex terms as predicate arguments and has restrictions on the use of negation and recursion. Thus, we decided not to use it. RuleML was also discarded, since it uses Datalog as basis for inferences. Rule Exchange Format (RIF) [Kifer 2008], on the other hand, is a W3C Recommendation to represent rules over the internet. It became a W3C recommendation in mid 2010. Since this is a very recent proposal, reasoners and other related technology are still under development. Consequently, we opted for using a Prolog inference machine to interpret implicit data and define new information over XML data. In the future, we may migrate to RIF.

The remaining of this work is organized in three sections. Section 2 illustrates our proposed approach, which we call XMLInference, and describes our XML-to-Prolog translation method. Section 3 discusses related work. Finally, section 4 concludes the article.

2. XMLINFERENCE: A QUERY METHOD TO INFER DATA

Our approach processes data contained in the XML structure with the goal of inferring new information. It enables more complex answers to be obtained from either explicit information or information deducted through inference rules (implicit information). For this to be possible, we build a knowledge base with facts (representing the XML data), rules (derived from the document's schema), and manual rules (provided by an expert user). This extends the query possibilities, providing the means for inference of new information.

Our method, called XMLInference, is divided in three stages: rules configuration, generation of facts, and query. Figure 1 illustrates each stage and its phases. During the rules configuration stage, the user informs to the system the schemas that validate the XML documents that will be queried. Such schemas are analyzed by a schema translator component that generates Prolog rules from the processed data. We call such rules "automatic rules". This component is also responsible for generating predicate definitions, which are Prolog predicate signatures. Using these definitions, an expert user can manually generate new rules. In the end of the first stage, the knowledge has Prolog rules that were automatically derived by our method, and manual rules that were added by the expert. Such knowledge base can be stored as a new project and can be loaded whenever needed.

The second stage uses the XML translator to derive Prolog facts from the XML documents. After this stage, all information contained in the XML document and in the schemas is defined in Prolog as facts and rules, respectively, together with manual rules. Such information constitutes the knowledge base and are subject to inference. Finally, in the third stage, the user is able to perform queries that are processed by the Prolog inference machine. The results are presented to the user that may submit additional queries, since the environment is all set up.

```
<customer>
  <history>
    <operation>buy</operation>
    performed in
    <date>02/23/1990</date>
    <amount currency="US$">586.00</amount>
  </history>
</customer>
```

```
1. customer(id1).
2. history(id1, id2).
3. operation(id2, 'buy').
4. xml/mixedElement(id2, 'performed in').
5. date(id2, '02/23/1990').
6. amount(id2, id3, '586.00').
7. currency(id3, 'US$').
```

Fig. 2. Example of XML document (left) and its Prolog translation (right)

Sections 2.1, 2.2 and 2.3 explain the translation rules of stages 1 and 2 of Figure 1. For didactical reasons, the translation of XML documents is explained before the schema translation. This inversion was intentional: since facts are simpler than rules, we present them beforehand, so that the reader gets familiarized with the syntax. In practice, this process is transparent to the user. This way, Section 2.1 explains the translation of XML documents to Prolog facts (second stage). Sections 2.2 and 2.3 explain the translation of the XML schemas to automatic and manual rules, respectively.

2.1 XML to Prolog translation

The translation process generates several facts for a single XML document. It transforms XML elements into predicates, and its content into constants. To relate distinct predicates, an identifier (Prolog constant) is added as a parameter. This allows to connect facts that represent parent/child relations in the XML document. The translation process is guided by five possible translation types, as follows. An example of the application of these translations is presented in Figure 2.

Translation of the document root (Figure 2, line 1). The XML document root must be treated in a special way, because this is the only element in the document that does not have a parent. In the large majority of the cases, the root is a complex element (unless it is a rare case of a document that contains a single simple element). Thus, the result of the translation of complex roots is a fact with the element name and a single argument that is equal to an application generated id. The main purpose of this id is to establish a connection with the child elements of the root.

Translation of complex elements (Figure 2, line 2). Complex elements have other elements as children. Thus, a new identifier is generated to relate the children with their parent. As the result of the translation, XMLInference generates a fact with the complex element name and two arguments: the parent's id and the element's id. Its children are translated accordingly to its type (simple, complex, etc).

Translation of atomic elements, with no attributes (Figure 2, lines 3 and 5). Atomic elements with no attributes have its textual content as its single child in the XML tree. Thus, they are translated as a Prolog fact named after the element's name. The arguments, in this case, are the parent's id and the element content.

Translation of atomic elements with attributes (Figure 2, lines 6 and 7). Attributes are considered children of the element that possesses them. We generate a new identifier for the element, so that its attribute child can reference it. XMLInference generates a fact that represents the element, plus a fact for each of its attributes. The former has the element name and three arguments: the parent's id, the new element's identifier, and its textual content. The remaining facts have the attribute name and two arguments: the parent's id (that represents the element that contains the attribute), and the attribute value.

Translation of mixed elements (Figure 2, line 4). Mixed elements are translated similarly to complex elements, except by the generation of "xml/mixedElement" facts to the element's text children (once they do not have names). This naming scheme was adopted considering that the XML specification does not allow XML element names to contain the "/" character. Thus, no other

```
<xs:element name="address" type="tAddress"/>
<xs:complexType name="tAddress">
  <xs:sequence>
    <xs:element name="street" type="xs:string"/>
    <xs:element name="house_number" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

```
address(ID, STREET, HOUSE_NUMBER) :-
   street(ID, STREET),
   house_number(ID, HOUSE_NUMBER).
```

Fig. 3. Example of XML Schema with "sequence" (left) and its Prolog translation (right)

translation can generate facts named "xml/mixedElement". This way, our translation mechanism generates "xml/mixedElement" facts with two arguments: the parent's id (the complex element) and its textual content. The other children are translated according to the appropriate translation rule.

2.2 Automatic Rules

Prolog rules are needed to make inference possible. Such rules are categorized into two types: manual and automatic rules. Automatic rules are automatically obtained from the document's XML Schema. They define the structure that must be followed by the facts in the knowledge base. Manual rules are inserted by an expert with the goal of increasing the inference power of our approach. In this section, we focus on automatic rules. Section 2.3 explains manual rules.

Our translation method generates different Prolog rules for XML Schema that validates the XML documents loaded into our knowledge base. This translation considers all three XML Schema group delimiters: *sequence*, *choice* and *all* [Fallsite and Ghemawat 2004]. These group delimiters define the structure of the XML documents. The next sections present the translation method that generates the automatic rules (stage 1 of our approach).

2.2.1 *Complex Elements with Simple Children.* The simplest cases of automatic rules are derived from complex elements that possess only simple elements as children. In this case, basic translation rules for each of the group delimiters *sequence*, *choice*, and *all* are used.

Translation of sequence. Among the existing group delimiters, the most used is *sequence* [Laender et al. 2009]. Its specification suggests that the complex element using it must follow the structure defined at the schema, respecting order and cardinality. The translation method that treats the schema produces a rule that respects the structure established by the analyzed *sequence*. Figure 3 (left side) shows part of an XML schema containing a *sequence* group delimiter.

The head of the Prolog rule generated in this case refers to the complex element ("address" in Figure 3) and has the following arguments: a Prolog variable representing the rule's identifier "ID", responsible for relating the rule to the other terms defined in Prolog (facts and rules), and one variable to each of the complex element's children ("STREET" and "HOUSE_NUMBER" in our example), in order to obtain the value of these attributes through the association of the identifier with the facts (or rules) referring to each of them. The tail of the rule possesses a fact referring to each child, with the same arguments of the rule's identifier (establishing its relation with the head) and the corresponding variable. Figure 3 shows in the right-hand side the automatic rule obtained from the translation of the schema presented in the left-hand side of Figure 3.

Translation of choice. The specification of the *choice* group delimiter allows a complex element to have as its child just one of the options defined in its structure. Thus, the translation method creates a rule for each option of *choice*, attending each of the possible valid structures. Figure 4 shows part of a schema with the occurrence of the *choice* group delimiter.

The result of the translation is a set of Prolog rules referring to the possible options of the structure that the complex element might have, as illustrated in Figure 4, right-hand side. The head of rule is formed by the complex element "unit", having as arguments a variable representing the rule's identifier

```
<xs:element name="unit" type="tUnit"/>
<xs:complexType name="tUnit">
  <xs:choice>
    <xs:element name="area" type="xs:string"/>
    <xs:element name="volume" type="xs:string"/>
  </xs:choice>
</xs:complexType>
```

```
unit(ID, area, AREA) :-
   area(ID, AREA).

unit(ID, volume, VOLUME) :-
   volume(ID, VOLUME).
```

Fig. 4. Example of XML Schema with "choice" (left) and its Prolog translation (right)

```
<xs:element name="contact" type="tContact"/>
<xs:complexType name="tContact">
  <xs:all>
    <xs:element name="phone" type="xs:string"/>
    <xs:element name="email" type="xs:string"/>
  </xs:all>
</xs:complexType>
```

```
contact(ID, phone, PHONE) :-
  phone(ID, PHONE).

contact(ID, email, EMAIL) :-
  email(ID, EMAIL).
```

Fig. 5. Example of XML Schema with "all" (left) and its Prolog translation (right)

```
<contact>
   <email>email@email.com</email>
   <phone>77777777</phone>
</contact>
```

Fig. 6. Example of XML document according to the schema of Figure 5

"ID", an option identifier (we use the element's name), responsible for indicating which of the options was adopted, and a variable referring to the corresponding element. In Figure 4, "area" and "volume" are respectively the identifiers of the options, while "AREA" and "VOLUME" are the variables. The tail of the rule has the fact referring to the child corresponding to the rule with two arguments: the rule's identifier (to establish the relation with the rule) and the appropriate variable.

Translation of all. The group delimiter *all* allows the complex element to have any combination of the options defined in its structure (in any order), making the occurrence of several different elements possible. During the translation, the element that has the group delimiter turns out to be the rule's head, which will be defined with the following parameters: a rule identifier "ID", an identifier of the option and a variable that represents the child corresponding to the chosen option. Once more, the rule will relate the possible children through its ID. Figure 5 (left-hand side) shows part of a schema that has a complex element *all*, whereas Figure 5 (right-hand side) shows the result of the translation of this same part of schema using the proposed algorithm.

The specification of the group delimiter *all* defines that it can only be used as the most external group of any complex type, and also that its children should all be elements. So there cannot be complex structures contained in or that contains the group delimiter *all*. Another restriction prevents the children elements of *all* to have cardinality greater then 1, limiting to 0 or 1 the values allowed for *minOccurs* and indicating that the value of *maxOccurs* should be 1.

In the right-hand side of Figure 5, "phone" is the option identifier of the first generated rule that refers to the *phone* element. Notice that *phone* is a child of the complex element *all*. On the other hand, "PHONE" is a variable that represents the textual content of the *phone* element. It is important to note that the option identifier acts as metadata of *contact*, while the variables represent the values themselves. This enables queries such as: `contact(7, METADATA, DATA).`

Assuming the XML document presented at Figure 6 and assuming that "contact" was generated with ID=7, the aforementioned query would generate the following result.

```
METADATA = email, DATA = email@email.com;
METADATA = phone, DATA = 77777777;
```

```
<xs:element name="customer" type="tCustomer"/>
<xs:complexType name=" tCustomer ">
 <xs:sequence>
   <xs:choice>
     <xs:element name="ssn" type="xs:string"/>
     <xs:element name="company_id" type="xs:string"/>
   </xs:choice>
   <xs:element name="phone" type="xs:string"/>
 </xs:sequence>
</xs:complexType>
```

```
customer(ID, ssn, SSN, PHONE) :-
  ssn(ID, SSN),
  phone(ID, PHONE).

customer(ID, company_id, COMPANY_ID, PHONE) :-
  company_id(ID, COMPANY_ID),
  phone(ID, PHONE).
```

Fig. 7. Sequence with a choice child (left) and its Prolog translation (right)

```
<xs:element name="customer" type="tCustomer"/>
<xs:complexType name="tCustomer">
  <xs:choice>
    <xs:sequence>
      <xs:element name="company_name" type="xs:string"/>
      <xs:element name="company_id" type="xs:string"/>
      <xs:element name="ssn_owner" type="xs:string"/>
    </xs:sequence>
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="ssn" type="xs:string"/>
    </xs:sequence>
  </xs:choice>
</xs:complexType>
```

```
customer(ID, s1, COMPANY_NAME,
         COMPANY_ID, SSN_OWNER) :-
      company_name(ID, COMPANY_NAME),
      company_id(ID,COMPANY_ID),
      ssn_owner(ID, SSN_OWNER).

customer(ID, s2, NAME, SSN) :-
      name(ID, NAME), ssn(ID, SSN).
```

Fig. 8. Choice with a sequence child (left) and its Prolog translation (right)

2.2.2 *Complex Elements with Complex Children.* The translation method for complex elements with complex elements as children adopts the same principles of the previously mentioned basic translation rules, used to generate automatic rules. However, because children elements are complex, some additional care is needed to precisely map the valid structures of the XML document into rules.

Translation of sequence with choice child. The translation of a group delimiter *sequence*, which owns a *choice* child, adopts rules defined as the union of the translation of both groups. The schema scrap presented in Figure 7 shows that the complex element "customer" has two children: the first may be "ssn" or "company_id" and the second is always "phone". This results in a situation similar to the existence of two different *sequences*, one of them containing "ssn" and "phone" and the other containing "company_id" and "phone". This way, the translation consists of two distinct rules, one for each situation. It is important to notice that the head of the rule also contains the choice to facilitate future queries. The right-hand side of Figure 7 shows the results of the translation of the schema shown in the left-hand side of Figure 7.

Translation of choice with sequence child. A complex element delimited by *choice* may have *sequence* children. If this happens, our translation method adopts a Prolog constant $(s1, ..., sn)$ for each *sequence* child. These constants work as identifiers of the possible choices. This is necessary because the *sequence* group delimiter has no corresponding name that can be used to identify the Prolog rule. The remaining of the process is similar to the translation of *choice* with simple children elements. Figure 8 illustrates the translation process.

There are also other possible combinations of group delimiters, and we treat them by applying the "neutral element property", as explained next.

The neutral element property. If a complex element has the same group delimiter as its parent (*sequence* with *sequence* child, or *choice* with *choice* parent), then we can treat it as a neutral element in our translation. To understand this concept, let C be the *choice* group delimiter, let S be the

```
<xs:element name="customer" type="tCustomer"/>
<xs:complexType name=" tCustomer">
  <xs:sequence>
    <xs:sequence>
      <xs:element name="name"  type="xs:string"/>
     <xs:element name="ssn"  type="xs:string"/>
    </xs:sequence>
    <xs:element name="phone" type="xs:string"/>
  </xs: sequence >
</xs:complexType>
```

```
<xs:element name="customer" type="tCustomer"/>
<xs:complexType name="tCustomer">
  <xs:sequence>
    <xs:element name="name" type="xs:string"/>
    <xs:element name="ssn" type="xs:string"/>
    <xs:element name="phone" type="xs:string"/>
  </xs: sequence >
</xs:complexType>
```

```
customer(ID, NAME, SSN, PHONE) :- name(ID, NAME), ssn(ID, SSN), phone(ID, PHONE).
```

Fig. 9. XML Schema with nested *sequence* group delimiters (left), simplified schema (right), and its Prolog translation

```
<xs:element name="unit" type="tUnit"/>
<xs:complexType name="tUnit">
  <xs:choice>
    <xs:element name="area" type="xs:string"/>
    <xs:element name="lenght" type="xs:string"/>
    <xs:choice>
      <xs:element name="mass" type="xs:string"/>
      <xs:element name="volume" type="xs:string"/>
    </xs:choice>
  </xs:choice>
</xs:complexType>
```

```
<xs:element name="unit" type="tUnit"/>
<xs:complexType name="tUnit">
  <xs:choice>
    <xs:element name="area" type="xs:string"/>
    <xs:element name="length" type="xs:string"/>
    <xs:element name="mass" type="xs:string"/>
    <xs:element name="volume" type="xs:string"/>
  </xs:choice>
</xs:complexType>
```

```
unit(ID, area, AREA) :- area(ID, AREA).
unit(ID, length, LENGTH) :- length(ID, LENGTH).
unit(ID, mass, MASS) :- mass(ID, MASS).
unit(ID, volume, VOLUME) :- volume(ID, VOLUME).
```

Fig. 10. XML Schema with nested *choice* group delimiters (left), simplified schema (right), and its Prolog translation

sequence group delimiter, and let X, Y, and Z be child elements of C or S.

The *sequence* specification defines that its children must appear in the XML document in the same order they appear in the schema. Thus, a *sequence* that has a *sequence* child preserves this restriction. This allows the following transformation: $S(X, S(Y, Z)) = S(X, Y, Z)$. We make this simplification in the schema before running our translation algorithm, as shown in Figure 9.

The same reasoning applies to the *choice* group delimiter. The *choice* specification requires that the XML document contain only one of the elements defined inside it in the schema. In the case a *choice* has another *choice* as a child, it can be interpreted as a single choice that includes the child elements of both. Formally, we have $C(X, C(Y, Z)) = C(X, Y, Z)$. Figure 10 illustrates such case.

There is also a special treatment for optional elements and for elements with cardinality greater than one. We omit the translation details due to space restrictions.

It is worth noting that we do not adopt any existing method to translate XML to Prolog [Coelho and Florido 2003; Seipel 2002; Wielemaker 2005] because these methods work at a coarse grain, mapping the whole XML into one Prolog fact. We need a fine-grained mapping, where elements become Prolog facts. This unleashes the Prolog inference machine to use features like backtracking when answering complex queries.

Moreover, the main goal of automatic rules is to ease the process of writing manual rules. This occurs because automatic rules are adherent to the structure of the document, and try to simplify the usage of basic Prolog facts generated from the document. Thus, our approach can be adopted even if the document does not have an associated schema. However, in this situation, no automatic rule can be generated and probably more manual rules will have to be created.

```
<xs:element name="entity" type="tEntity"/>
<xs:complexType name="tEntity">
  <xs:choice>
    <xs:sequence>
      <xs:element name="company_name" type="xs:string"/>
      <xs:element name="company_ID" type="xs:string"/>
      <xs:element name="ownerSSN" type="xs:string"/>
    </xs:sequence>
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="ssn" type="xs:string"/>
      <xs:element name="parentName" type="xs:string"/>
    </xs:sequence>
  </xs:choice>
</xs:complexType>
```

```
partner(Name1, Name2) :-
   name(Person1, Name1),
   ssn(Person1, SSN1),
   ownerSSN (Company1, SSN1),
   company_id(Company1, CompanyID),
   name(Person2, Name2),
   ssn(Person2, SSN2),
   ownerSSN (Company2, SSN2),
   company_id(Company2, CompanyID),
   Name1 \= Name2.
```

Fig. 11. Example of XML Schema and a manual Prolog rule

```
for $person1 in doc("test2XML.xml")//entity, $person2 in doc("test2XML.xml")//entity,
    $company1 in doc("test2XML.xml")//entity, $company2 in doc("test2XML.xml")//entity
where $person1/ssn = $company1/ownerSSN and $person1/ssn != $person2/ssn
  and $person2/ssn = $company2/ownerSSN and $company1/company_id = $company2/company_id
return {$person1/name/text()} is a partner of {$person2/name/text()}
```

Fig. 12. XQuery about customers and partnership

2.3 Manual Rules

Manual rules are elaborated rules that cannot be automatically obtained from the schema. The approach we propose allows the insertion of manual rules by an expert user during the rules configuration stage. The goal is to enable more complex inferences, thus achieving query results that cannot be obtained from the information contained in the document or in the XML schema.

As an example, the schema defined in Figure 11 does not have any information regarding partnership. Nevertheless, it might be necessary to establish this relation among the registered people. Analyzing the definition of the predicates generated from the schema translation, an expert user (with domain knowledge and Prolog knowledge) can create rules that make it possible to infer such information. The semantics behind a set of XML tags (for example, the fact that two owners of the same business are partners) is not automatically captured by a translation schema, but it can be noticed by knowledge engineers (expert users). They can evaluate if this concept enriches the knowledge base and decide to include it in the form of a rule. It is worth mentioning that this only occurs at the configuration stage, and that these users might not be the ones that will use the query system later (the experts might be there during the environment setup only).

Figure 11, right side, illustrates the manual rule that must be inserted by an expert user to implement the partnership relation among people. During the query stage, the user gets information about the partnership relation through the query term: "partner(X, Y)". With XQuery, the same information would be obtained by means of a much more complex query, shown in Figure 12.

The inference mechanism from Prolog allows the reuse of rules to elaborate more complex queries. Considering our example, Figure 13 shows four Prolog rules that make it possible to obtain query results with respect to partnerships originated from inheritance (heir partners). Notice that the rules "heirPartner" reuse previously defined rules "partner" and "heir". This is not allowed in XQuery: XQuery users must define all the desired relations in each query.

```
heir(Heir, Predecessor) :- name(ID, Heir), parentName(ID, Predecessor).
heir(Heir, Predecessor) :- name(ID, Heir), parentName(ID, Predecessor2),
                           heir(Predecessor2, Predecessor).
heirPartner(Name1, Name2) :- hairPartner2(Name1, Name2).
heirPartner(Name1, Name2) :- hairPartner2(Name2, Name1).
heirPartner2(Name1, Name2) :- heir(Name1, Predecessor1), partner(Predecessor1,Name2).
heirPartner2(Name1, Name2) :- heir(Name1, Predecessor1), heir(Name2, Predecessor2),
                              partner(Predecessor1, Predecessor2).
```

Fig. 13. Example of heirPartner manual rule

3. RELATED WORK

The integration of Logic Programming and the Database field has been studied with the intention to ease the representation and manipulation of data. [Niemi and Jarvelin 1991] proposed the adoption of Prolog to represent a relational database's knowledge base. With the dissemination of the Internet and the emergence of large amounts of XML data, the interest in the integration of Logic Programming with the processing of XML data has grown [Boley 2000; Almendros-Jiménez et al. 2008; Bailey et al. 2005; Seipel 2002].

Similar to us, Almendros-Jiménez et al. (2008) propose a translation from XML documents into facts, and from schemas into Prolog rules. The goal of their work is to implement the XPath language in Logic Programming. Their method, tough, translates only the necessary path to answer a submitted query, whereas our work translates the whole document, allows the insertion of new rules and the use of Prolog queries.

Seipel (2002) proposes a model to represent XML documents in Prolog called field-notation. The main goal of his work is to show that semi-structured data can be represented and queried using logic programming. His translation, however, differs from the one we propose here once Seipel uses association lists to represent data in Prolog. Thus, his method generates a list containing the entire XML document. Elements inside this list are nested and represented as attribute/value pairs.

In his work, Boley (2000) establishes a strong relation between logic programming and XML. His goal is to investigate the use of XML in formal documents. Thus, he investigates XML to prolog translation, and also Prolog to XML translation. He represents Herbrand terms and Horn clauses in an XML language called XmlLog. He then investigates how to query the resulting XML document by using XQL. Once more, his translation differs from ours in the sense that it generates large Prolog facts, which overcomplicates the query process.

Coelho and Florido (2003) make types' inference in XML documents with the usage of logic programming. This approach aims at deducing the data type of elements contained in the structure and use a DTD to support the translation process. In our approach, we adopt the use of XML Schema. As we propose, Coelho and Florido use Prolog to make inferences over XML data, but their work is concerned with the inference of data types instead of content. Similar to the work of Boley (2000) and Seipel (2002), his translation method also generates large Prolog facts.

4. FINAL REMARKS

This work proposes a query method for XML documents that is not restricted only to the documents' explicit data. With the support of previously loaded XML Schemas, rules that should be followed by the instances (database facts) can also be queried and used for the inference of implicit information, enriching the queries' results.

Even though the initial information set is represented in XML documents, the users should make their queries in Prolog, once all data is translated to this language. This is done in order to make inference possible. In spite of the fact that many of the users interested in querying XML documents have a technological background and could possibly be familiarized with Prolog, this obstacle can be eased with the adoption of a graphical interface that generates the Prolog queries from the selection of high level options [Costa and Braganholo 2010].

A limitation of our approach resides in the fact that Prolog treats upper-case text as variables. Thus, our translation mechanism for *choice* and *all* group delimiters does not work when the XML element is written in uppercase. A possible solution to this limitation is to pre-process the document and its schema, translating all element names to lowercase. Another one is to use an escape character in Prolog. We are studying alternative solutions to this limitation.

We are currently working on the experimental evaluation of our approach. It will be based on the XBench XML benchmark [Yao et al. 2004]. In this experiment, we plan to measure both expressiveness and query processing time. As future work, we intend to experiment with users with different backgrounds and expertise levels, to evaluate the usability our approach from the user's point of view.

REFERENCES

Almendros-Jiménez, J. M., Becerra-Terón, A., and Enciso-Baños, F. J. Querying XML documents in logic programming. *Journal of Theory and Practice of Logic Programming* 8 (3): 323–361, 2008.

Bailey, J., Bry, F., Furche, T., and Schaffert, S. Web and semantic web query languages: a survey. In *Reasoning Web*, N. Eisinger and J. Maluszynski (Eds.). Lecture Notes in Computer Science, vol. 3564. Springer, pp. 35–133, 2005.

Boag, S., Chamberlin, D., Fernandez, M., Florescu, D., Robie, J., and Simeon, J. XQuery 1.0: an XML query language (Second Edition). *W3C Recommendation*, 2010. Available at http://www.w3.org/TR/xquery/.

Boley, H. Relationships between logic programming and XML. In *Workshop on Logic Programming*. Berlin, Germany, pp. 19–34, 2000.

Boley, H. The rule markup language: RDF-XML data model, XML schema hierarchy, and XSL transformations. In *International Conference on Web Knowledge Management and Decision Support*. Tokyo, Japan, pp. 5–22, 2001.

Clark, J. and Derose, S. XML Path Language (XPath) Version 1.0. *W3C Recommendation*, 1999. Available at http://www.w3.org/TR/xpath/.

Coelho, J. and Florido, M. Type-based XML processing in logic programming. In *International Symposium on Practical Aspects of Declarative Languages*. New Orleans, USA, pp. 273–285, 2003.

Costa, P. V. and Braganholo, V. Uma nova abordagem para consulta a dados de proveniência. In *Workshop de Teses e Dissertações em Banco de Dados (WTDBD)*. Belo Horizonte, MG, pp. 1–6, 2010.

Fallsite, D. and Ghemawat, S. XML Schema Part 0: Primer. Second Edition. *W3C Recommendation*, 2004. Available at http://www.w3.org/TR/xmlschema-0/.

Freire, J., Koop, D., Santos, E., and Silva, C. T. Provenance for computational tasks: A survey. *Computing in Science and Engineering* 10 (3): 11–21, 2008.

Gallaire, H. and Minker, J. *Logic and data bases*. New York: Plenum, 1978.

Kifer, M. Rule interchange format: the framework. In *International Conference on Web Reasoning and Rule Systems*. Karlsruhe, Germany, pp. 1–11, 2008.

Kotsakis, E. Structured information retrieval in XML documents. In *ACM Symposium on Applied Computing*. Madrid, Spain, pp. 663–667, 2002.

Laender, A. H., Moro, M. M., Nascimento, C., and Martins, P. An x-ray on web-available XML schemas. *ACM SIGMOD Record* 38 (1): 37–42, 2009.

Lloyd, J. *Foundations of logic programming*. New York: Springer-Verlag, 1984.

Mattoso, M., Werner, C., Travassos, G. H., Braganholo, V., Murta, L., Ogasawara, E., Oliveira, D., Cruz, S. M. S. d., and Martinho, W. Towards supporting the life cycle of large-scale scientific experiments. *International Journal of Business Process Integration and Management* 5 (1): 79–92, 2010.

Niemi, T. and Jarvelin, K. Prolog-based meta-rules for relational database representation and manipulation. *IEEE Transactions on Software Engineering* 17 (8): 762–788, 1991.

Seipel, D. Processing XML-documents in prolog. In *Workshop Logische Programmierung*. Dresden, Germany, pp. 1–15, 2002.

Wielemaker, J. SWI-Prolog SGML/XML Parser, Version 2.0.5. Tech. rep., Human Computer-Studies (HCS), University of Amsterdam, 2005.

Yao, B., Ozsu, T., and Khandelwal, N. XBench Benchmark and Performance Testing of XML DBMSs. In *International Conference on Data Engineering*. Boston, MA, USA, pp. 621–632, 2004.