Querying XML documents in logic programming*

J. M. ALMENDROS-JIMÉNEZ, A. BECERRA-TERÓN
and F. J. ENCISO-BAÑOS

Dpto. de Lenguajes y Computación, Universidad de Almería
(*e-mail:* `{jalmen,abecerra,fjenciso}@ual.es`)

Abstract

Extensible Markup Language (XML) is a simple, very flexible text format derived from SGML. Originally designed to meet the challenges of large-scale electronic publishing, XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere. *XPath* language is the result of an effort to provide address parts of an XML document. In support of this primary purpose, it becomes in a *query language* against an XML document. In this paper we present a proposal for the implementation of the *XPath* language in logic programming. With this aim we will describe the representation of XML documents by means of a *logic program*. *Rules* and *facts* can be used for representing the document schema and the XML document itself. In particular, we will present *how to index XML documents* in logic programs: rules are supposed to be stored in *main memory*, however facts are stored in *secondary memory* by using two kind of indexes: one for each XML tag, and other for each group of terminal items. In addition, we will study how to query by means of the *XPath* language against a logic program representing an XML document. It evolves the *specialization of the logic program* with regard to the *XPath* expression. Finally, we will also explain how to *combine the indexing and the top-down evaluation* of the logic program.

KEYWORDS: Logic Programming, XML, XPath.

1 Introduction

Extensible Markup Language (XML) (W3C 2007a) is a simple, very flexible text format derived from SGML. Originally designed to meet the challenges of large-scale electronic publishing, XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere.

XPath language (W3C 2007b) is the result of an effort to provide address parts of an XML document. In support of this primary purpose, it becomes in a query language against an XML document, providing basic facilities for manipulation of

strings, numbers and booleans. *XPath* uses a compact, non-XML syntax to facilitate the use of *XPath* within *URIs* and XML attribute values. *XPath* operates on the abstract, logical structure of an XML document, rather than its surface syntax. *XPath* gets its name from its use of a path notation as in *URLs* for navigating through the hierarchical structure of an XML document.

Essential to *semi-structured data* (Abiteboul *et al.* 2000) is the selection of data from incompletely specified data items as in an XML document. For such data selection, the *XPath* language is a path language which provides constructors similar to regular expressions and *"wildcards"* allowing a flexible node retrieval. The *XML schema* (W3C 2001), which is also an XML document, defines the structure of well-formed documents and thus it can be seen as a type definition.

The integration of *logic programming languages* and *web technologies*, in particular XML data processing, is interesting from the point of view of the applicability of logic programming.

On one hand, XML documents are the standard format of *exchanging information between applications*, therefore logic languages should be able to handle and query such documents.

On the other hand, logic languages could be used for *extracting and inferring semantic information* from XML documents, in the line of *"Semantic Web"* requirements (Berners-Lee *et al.* 2001). Therefore logic languages can find a natural and interesting application field in this area.

1.1 Contributions of this paper

In this paper, we are interested in the use of logic programming for handling XML documents and *XPath* queries. In this context, our contributions can be summarized as follows:

1. An XML document can be seen as a logic program by considering *facts* and *rules* for expressing both the XML schema and document.

   On one hand, rules can describe the *schema of an XML document* in which a (possibly recursive) definition specifies the well-formed documents.

   On the other hand, each XML document can be described by means of facts, one for each terminal item (i.e. the XML tree leaves). Although the XML schema is usually available for XML documents, our method has been studied for extracting the XML schema from the XML document itself. It can be considered in a certain sense as a type inference. As future work, we will consider to adapt our technique to directly translate XML schemas into logic rules.

2. Our second contribution is the following: once XML documents can be described by means of a logic program, an *XPath* expression against the document requires to obtain a subset of the *Herbrand model* (Apt 1990) represented by the logic program. In other words, only a subset of the facts representing the XML document is required for each *XPath* query.

   Our idea is to provide a *specialization program method* in order to retrieve only the subset of the Herbrand model required for answering the query. In other words, we will specialize the logic program representing an XML document with

regard to an *XPath* expression in order to get the answer; that is, the XML data relevant to the query.

Basically, the *specialization technique* will consist on *specialization of rules* by *removing and reordering predicates*. It will be achieved on the rules for the schema of the XML document, which now can be used for retrieving a subset of the set of facts representing the XML document. In addition, for each *XPath* query, a specific goal (or goals) is called, where *appropriate arguments can be instantiated*. It depends on the occurrences of boolean conditions in the *XPath* expression.

3. Our technique allows the handling of XML documents as follows.

Firstly, the XML document is loaded. It involves the translation of the XML document into a logic program. For efficiency reasons, the rules corresponding to the XML schema are loaded in *main memory*, but facts, which basically represent the XML document, are stored in *secondary memory* (using appropriate *indexing techniques*) whenever they do not fit in main memory.

Secondly, the user can now write queries against the loaded document. For query solving the logic program (corresponding to the XML schema) is specialized for each query, and the top-down evaluation of such specialized program computes the answer. The indexing technique allows that the query solving is *more efficient*, that is, it uses indexes for retrieving the facts required for the answer.

4. We have developed a prototype called *XIndalog* which implements *XPath* following the technique presented in this paper. This prototype is hosted at `http://indalog.ual.es/XIndalog` in order to be tested.

We have tested our prototype with not enough structured documents and complex queries, and with big documents of different sizes. We will show benchmarks of our prototype, comparing answer times with and without our specialization technique.

Our approach opens two promising research lines.

- The first one, the extension of *XPath* to a more powerful query language such as *XQuery* (W3C 2007c; Chamberlin *et al.* 2004; Wadler 2002; Chamberlin 2002; Simeon and Wadler 2003; Fernández *et al.* 2000), that is, the study of the implementation of *XQuery* in logic programming.

  The current implementations of *XQuery* are implemented using as host language a functional language (see the *Galax* project (Chamberlin *et al.* 2004; Fernández and Simeon 2003; Marian and Simeon 2003)).

- The second one, the use of logic programming as *inference engine* for the so-called *"Semantic Web"* (Berners-Lee *et al.* 2001; Decker *et al.* 2000), by introducing semantic information like *RDF* (*Resource Description Framework*) documents (W3C 2004b) or *OWL* (*Ontology Web Language*) specifications (W3C 2004a) in the line of (Wolz 2004; Grosof *et al.* 2003; Horrocks and Patel-Schneider 2004).

1.2 Related Work

The integration of *declarative programming* and *XML data processing* is a research field of increasing interest in the last years. There are proposals of new languages for XML data processing based on functional, and logic programming (see (Bailey *et al.* 2005) for a survey). In addition, *XPath* and *XQuery* have been also implemented in declarative languages.

The most relevant contribution is the *Galax* project (Marian and Simeon 2003; Chamberlin *et al.* 2004), which is an implementation of *XQuery* in functional programming, using *OCAML*(Rémy 2002) as host language. There are also proposals for new languages based on functional programming rather than implementing *XPath* and *XQuery*. This is the case of *XDuce* (Hosoya and Pierce 2003) and *CDuce* (Benzaken *et al.* 2005), which are languages for XML data processing, using regular expression pattern matching over XML trees, subtyping as basic mechanism, and *OCAML* as host language. The *CDuce* language does fully statically-typed transformation of XML documents, thus guaranteeing correctness. In addition, there are proposals around *Haskell* for the handling of XML documents, such as *HaXML* (Thiemann 2002; Atanassow *et al.* 2004) and (Wallace and Runciman 1999).

There are also contributions in the field of logic programming for the handling of XML documents. For instance, the *Xcerpt* project (Schaffert and Bry 2002; Bry and Schaffert 2002a) proposes a pattern and rule-based query language for XML documents, using the so-called query terms including logic variables for the retrieval of XML elements. For this new language a specialized unification algorithm for query terms has been studied in (Bry and Schaffert 2002b). Another contribution of a new language is *XPathLog* (the *Lopix* system) (May 2004) which is a *Datalog*-style extension for *XPath* with variable bindings. *Elog* (Baumgartner *et al.* 2001) is also a logic-based XML data manipulation language, which has been used for representing Web documents by means of logic programming. This is also the case of *XCentric* (Coelho and Florido 2003, 2004), which can represent XML documents by means of logic programming, and handles XML documents by considering terms with functions of flexible arity and regular types. Finally, *FNPath* (Seipel 2002) is a proposal in order to use *Prolog* as query language for XML documents based on a field-notation, for evaluating *XPath* expressions based on *DOM*.

The *Rule Markup Language* (*RuleML*) (Boley 2001, 2000a,b) is a different kind of proposal in this research area. The aim of *RuleML* is the representation of *Prolog* facts and rules in XML documents, and thus, the introduction of *rule systems* into the *Web*.

Finally, some well-known *Prolog* implementations include libraries for loading and querying XML documents, such as *SWI-Prolog* (Wielemaker 2005) and *CIAO* (Cabeza and Hermenegildo 2001).

In the cited logic approaches interested in *XPath* queries (Schaffert and Bry 2002; May 2004) *XPath* is directly handled, that is, rules and queries use a new kind of *Prolog* terms adapted to XML patterns. It involves to study new unification algorithms for the new *Prolog* terms. However, in our work we will show how to handle XML documents not introducing new *Prolog* terms, but using the standard *Prolog* terms. In addition, in our case, *XPath* queries evolve a

program transformation. The top-down evaluation of the goals w.r.t. the transformed program obtains a set of answers which represents a subset of the Herbrand model of the transformed program. This subset allows the reconstruction of the XML document representing the answer. The reconstruction follows the same criteria as the translation of XML document-logic program.

Our proposal requires the representation of XML documents into logic programming, and thus it can be compared with those ones representing XML documents in logic programming for instance, (Schaffert and Bry 2002; Coelho and Florido 2003; Cabeza and Hermenegildo 2001; Wielemaker 2005) and, with those ones representing XML documents in relational databases for instance, (Boncz *et al.* 2005; O'Neil *et al.* 2004; Tatarinov *et al.* 2002). In our case, rules are used for expressing the structure of well-formed XML documents, and XML elements are represented by means of facts. Moreover, our handling of XML documents is more *"database-oriented"* since we use secondary memory and file indexing in order to retrieve the database records. The reason for such decision is that XML documents can usually be too big for main memory (Marian and Simeon 2003).

With regard to *RuleML* (Boley 2001), we translate XML documents into a logic program using facts and rules; however we are not still interested in the translation of logic rules into XML (or RDF) documents. This translation would be interesting when semantic information is handled by means of logic programming. In fact, our idea is to consider these aspects as future work in the line of (Wolz 2004; Grosof *et al.* 2003; Horrocks and Patel-Schneider 2004).

There is an analogy among our specialization technique and the *magic sets*-based program specialization technique used for deductive databases, which uses the *bottom-up* evaluation for answering queries. We have also studied such technique for XML documents in a previous work (Almendros-Jiménez *et al.* 2006). In fact, we have developed two releases of *XIndalog*: one of them implements the top-down approach presented in this paper and the other one implements the bottom-up approach.

The main differences between the top-down and the bottom-up approaches are the program transformation technique and evaluation method of queries. In the second case, we use: (1) the *fix-point* operator in order to evaluate *XPath* queries, and (2) a *magic sets* based technique in order to specialize and evaluate the program. With respect to the transformation of XML documents into a logic program, let us remark that this one in both approaches is the same. However, the specialization technique is different, the technique of this paper is based on predicate removing and reordering, and the instantiation of the goals called in a top-down fashion.

1.3 Structure of the Paper

The structure of the paper is as follows. Section 2 will review basic concepts of XML documents and *XPath* queries; section 3 will study the translation of XML documents into *Prolog*; section 4 will present the program specialization technique applied to *XPath* queries; section 5 will prove theoretical results about our technique; section 6 will show the indexing technique over XML documents represented by means of logic programming and will explain the combination of

the indexing and program specialization techniques; section 7 will show the Web prototype developed under *SWI-Prolog* for the language *XPath* at the University of Almeria (`http://indalog.ual.es/Xindalog`), presenting benchmarks of our prototype; and finally, section 8 will conclude and present future work.

2 XML and XPath

An *XML document* basically is a *labeled tree* with inner nodes representing *composed or non-terminal items* and leaves representing *values or terminal items*. For instance, let us consider the following XML document which we will use in the paper as running example:

```
<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>
```

In the XML document, the tags are used for specifying a set of *books* described by means of author's names, the title and a review. Each *book* is qualified by means an attribute called *year*. For each element *book*, we have three grouped subelements *author*, *title* and *review*. In addition, the element *review* contains subelements used for formatting the text described by the review.

Here, the XML database includes two books. The first one, edited in *2003*, with *author*s *Abiteboul*, *Buneman* and *Suciu*, and *title* "Data on the Web." Finally, the opinion of the reviewer for this book was: *"A fine book."* The second one, edited in *2002*, was written by *Buneman* with title *XML in Scotland*, and the opinion of the reviewer was *"The best ever!"*.

XML documents describe data by means of a *semi-structured data model* (Abiteboul *et al.* 2000), whose main features are the occurrences of *heterogeneous records*, and in particular, *non-first normal relations*, *missing values*, among others.

Now, with respect to the above XML document, we can consider the following two *XPath* expressions, as well as the expected answers in XML format:

XPath Expression	Expected XML Answer
(1) /books/book[author="Suciu"]/title	**(1)** <title>Data on the Web</title>
(2) /books//title	**(2)** <title>Data on the Web</title> **(2)** <title>XML in Scotland</title>

where **(1)** requests *Suciu*'s book titles, and **(2)** requests book titles without taking into account the structure of the book records.

3 Translating XML Documents into Logic Programming

In this section, we will show how to translate an XML document into a logic program. We will use a set of rules for describing the XML schema and a set of facts for storing the XML document.

In general, an XML document includes:

(a) *tagged elements* which have the form:

$$< tag \; att_1 = v_1, \ldots, att_n = v_n > subelem_1, \ldots, subelem_k < /tag >$$

where $att_1, \ldots, att_n$ are the attributes names, $v_1, \ldots, v_n$ are the attribute values supposed to have a *basic type*: strings, integers, real numbers, lists of integers or real numbers, and $subelem_1, \ldots, subelem_k$ are subelements; and

(b) *untagged elements* which have a basic type.

Terminal tagged elements (i.e. XML tree leaves) are those ones whose subelements have a basic type and do not have attributes. Otherwise they are called *non-terminal tagged elements* (i.e. inner nodes). Two tagged elements are *similar* whether they have the same structure; that is, they have the same tag and attributes names, and the subelements are similar. Untagged elements are always similar. Two tagged elements are *distinct* if they do not have the same tag and, finally, they are *weakly distinct* if they have the same tag but they are not similar.

3.1 Numbering XML documents

In order to define our translation we need to number the nodes of the XML document. Similar kinds of node numbering have been studied in some works about XML processing in relational databases (Boncz *et al.* 2005; O'Neil *et al.* 2004; Tatarinov *et al.* 2002). Our goal is similar to these approaches: to identify each inner node and leaf of the tree represented by the XML document.

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**[1] 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 $i_1.\ldots.i_t$:

$$< tag \; att_1 = v_1, \ldots, att_n = v_n, \textbf{nodenumber} = \textbf{i}_1.\ldots.\textbf{i}_t.\textbf{j} >$$
$$elem_1, \ldots, elem_s < /tag >$$

This is the case of tagged elements; If the j-th child has a basic type and the parent is a non-terminal tagged element then the element is labeled and numbered as follows:

$$< unlabeled \; \textbf{nodenumber} = \textbf{i}_1.\ldots.\textbf{i}_t.\textbf{j} > elem < /unlabeled >$$

---

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

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 on sequences of natural numbers. The node numbered XML document corresponding to the running example is as follows:

```
<books nodenumber=1>
<book year="2003", nodenumber=1.1>
<author nodenumber=1.1.1>Abiteboul</author>
<author nodenumber=1.1.2>Buneman</author>
<author nodenumber=1.1.3>Suciu</author>
<title nodenumber=1.1.4>Data on the Web</title>
<review nodenumber=1.1.5>
<unlabeled nodenumber=1.1.5.1> A </ unlabeled>
<em nodenumber=1.1.5.2>fine</em>
<unlabeled nodenumber=1.1.5.3> book. </ unlabeled>
</review>
</book>
<book year="2002" nodenumber=1.2>
<author nodenumber=1.2.1>Buneman</author>
<title nodenumber=1.2.2 >XML in Scotland</title>
<review nodenumber=1.2.3 >
<em nodenumber=1.2.3.1>
<unlabeled nodenumber=1.2.3.1.1> The </unlabeled>
<em nodenumber=1.2.3.1.2>best</em>
<unlabeled nodenumber=1.2.3.1.3> ever! </unlabeled>
</em>
</review>
</book>
</books>
```

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:

$$< tag \ att_1 = v_1, \ldots, att_n = v_n, nodenumber = i_1 \ldots, i_t.j, \textbf{typenumber} = \textbf{k} >$$
$$elem_1, \ldots, elem_s < /tag >$$

and

$$< unlabeled \ nodenumber = i_1 \ldots .i_t.j, \textbf{typenumber} = \textbf{k} >$$
$$elem < /unlabeled >$$

for "unlabeled" nodes. In both cases, the type number of the tag is $k = l + n + 1$ whenever the type number of the parent is l, and n is the number of tagged elements weakly distinct to the parent, occurring in leftmost positions at the same level of the XML tree. Therefore, all the children of a tag have the same type number.

For instance, with respect to the running example, we can see in the Figure 1 the type and node numbering which represent the following type and node numbered XML document.

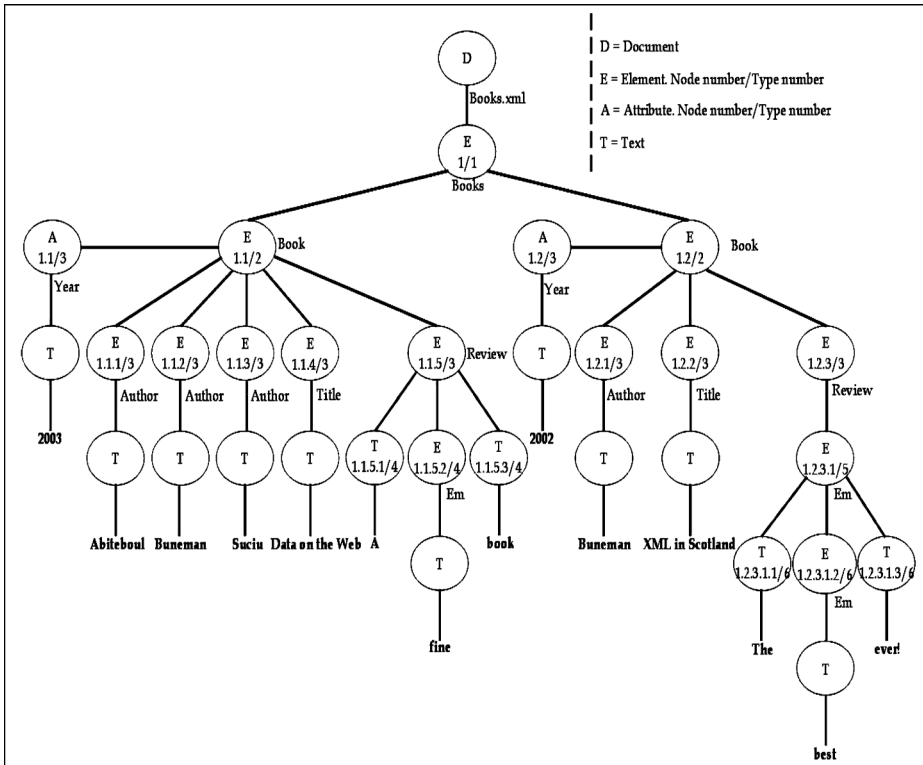Fig. 1. Type and node numbering in the XML tree of the running example.

```
<books nodenumber=1, typenumber=1>
<book year="2003", nodenumber=1.1, typenumber=2>
<author nodenumber=1.1.1 typenumber=3>Abiteboul</author>
<author nodenumber=1.1.2 typenumber=3>Buneman</author>
<author nodenumber=1.1.3 typenumber=3>Suciu</author>
<title nodenumber=1.1.4 typenumber=3>Data on the Web</title>
<review nodenumber=1.1.5 typenumber=3>
<unlabeled nodenumber=1.1.5.1 typenumber=4> A </ unlabeled>
<em nodenumber=1.1.5.2 typenumber=4>fine</em>
<unlabeled nodenumber=1.1.5.3 typenumber=4> book. </ unlabeled>
</review>
</book>
<book year="2002" nodenumber=1.2, typenumber=2>
<author nodenumber=1.2.1 typenumber=3>Buneman</author>
<title nodenumber=1.2.2 typenumber=3>XML in Scotland</title>
<review nodenumber=1.2.3 typenumber=3>
<em nodenumber=1.2.3.1 typenumber=5>
<unlabeled nodenumber=1.2.3.1.1, typenumber=6> The </unlabeled>
<em nodenumber=1.2.3.1.2, typenumber=6>best</em>
<unlabeled nodenumber=1.2.3.1.3,typenumber=6> ever! </unlabeled>
</em> </review>
</book>
</books>
```

Let us focus our attention to the type numbering of *review*. According to the proposed type numbering, the children of *review* are numbered as $k = l + n + 1$ where l is the type number of *review*, and n is the number of *weakly distinct records* of *review* at the same level of the tree. Therefore, the first set of children is numbered as $4 = 3 + 0 + 1$ and the second set of children is numbered as $5 = 3 + 1 + 1$ (i.e. the first and second reviews are weakly distinct). This kind of type numbering allows us to distinguish both kind of records and not to confuse them.

Let us remark that in practice the type and node numbering of XML documents can be simultaneously generated at the same time as the translation into the logic program. In fact, the type and node numbered version of the original XML document is not generated as an XML file.

3.2 Translation of XML documents

Now, the translation of the XML document into a logic program $\mathscr{P}$ is as follows. For each non-terminal tagged element 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*:

$$tag(tagtype(Tag_{i_1}, \ldots, Tag_{i_t}, \quad [Att_1, \ldots, Att_n]), NodeTag, k) :-$$
$$tag_{i_1}(Tag_{i_1}, [NodeTag_{i_1} | NodeTag], r),$$
$$\ldots,$$
$$tag_{i_t}(Tag_{i_t}, [NodeTag_{i_t} | NodeTag], r),$$
$$att_1(Att_1, NodeTag, r),$$
$$\ldots,$$
$$att_n(Att_n, NodeTag, r).$$

---

where

- *tagtype* is a new function symbol used for building a *Prolog* term containing the XML document;
- $\{tag_{i_j} \; | i_j \in \{1, \ldots, s\}, \; 1 \leqslant j \leqslant t\}$ 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;
- $Att_1, \ldots, At_n$ are variables, one for each attribute name;
- $NodeTag_{i_1}, \ldots, NodeTag_{i_t}$ are variables (used for representing the first digit of the node number of the children).
- $NodeTag$ is a variable (used for representing the node number of the tag).
- k is the type number of *tag*.
- r is the type number of the tagged elements in $elem_1, \ldots, elem_s$[2]

---

[2] Let us remark that given that *tag* is a tagged element then $elem_1, \ldots, elem_s$ have been tagged with "unlabeled" labels when they had a basic type in the type and node numbered XML document, and thus all of them have a type number.

In addition, we consider *facts* of the form:

$$att_j(v_j, i, k)$$

for each $1 \leqslant j \leqslant n$. Finally, for each terminal tagged element in the type and node numbered XML document:

$$< tag \ nodenumber = i, typenumber = k > value < /tag >$$

we consider the *fact*:

$$tag(value, i, k).$$

In summary, each non-terminal tag (element) is translated into a predicate name, with three arguments.

The first argument of the predicate is used for building a *Prolog* term containing the XML document. It consists of a function symbol named as "*elementname + type*" with an argument for each subelement and an additional argument for storing the list of attributes.

The second argument of the predicate is used for numbering each node of the XML document tree, and the third one is use for numbering each type.

Finally, each terminal element and attribute is translated into a fact.

Let us remark that the same "*elementname + type*" function symbol could have several occurrences with different arity depending on the document includes weakly distinct elements or not.

From a type and node numbered XML document $\mathcal{X}$, we can build a unique program $\mathcal{P}$, and conversely, from a logic program $\mathcal{P}$ we can build a unique type and node numbered XML document $\mathcal{X}$.

The logic program obtained from a document $\mathcal{X}$ is denoted by $Prog(\mathcal{X})$, and the XML document obtained from a program $\mathcal{P}$ is denoted by $Doc(\mathcal{P})$. In addition, $Doc(Prog(\mathcal{X})) = \mathcal{X}$ and $Prog(Doc(\mathcal{P})) = \mathcal{P}$.

Moreover, we can associate from our translation to each *tag* a set of patterns of the form $tagtype(\overline{Tag}, [\overline{Att}])$, denoted by $PT(tag)$.

Finally, to each *pattern t* of $PT(tag)$, we can associate the set of type numbers $\{r_1, \ldots, r_n\}$ assigned to t in our translation–there could be more than one type number for one pattern due to occurrences of weakly distinct elements. This set is denoted by $TN(t)$, and pattern instances $t\theta$ have the same set of type numbers, that is, $TN(t\theta) =_{def} TN(t)$ for all θ.

3.3 Examples

For instance, the running example can be represented by means of a logic program as follows:

Rules (Schema):

---

books(bookstype(Books, [ ]), NodeBooks,1) :-
        book(Books, [NodeBook|NodeBooks],2).
book(booktype(Author, Title, Review, [Year]), NodeBook ,2) :-
        author(Author, [NodeAuthor|NodeBook],3),
        title(Title, [NodeTitle|NodeBook],3),
        review(Review, [NodeReview|NodeBook],3),
        year(Year, NodeBook,3).
review(reviewtype(Unlabeled,Em,[ ]),NodeReview,3):-
        unlabeled(Unlabeled,[NodeUnlabeled|NodeReview],4),
        em(Em,[NodeEm|NodeReview],4).
review(reviewtype(Em,[ ]),NodeReview,3):-
        em(Em,[NodeEm|NodeReview],5).
em(emtype(Unlabeled,Em,[ ]),NodeEms,5) :-
        unlabeled(Unlabeled,[NodeUnlabeled|NodeEms],6),
        em(Em, [NodeEm|NodeEms],6).

Facts (Document):

---

year('2003', [1, 1], 3).
author('Abiteboul', [1, 1, 1], 3).
author('Buneman', [2, 1, 1], 3).
author('Suciu', [3, 1, 1], 3).
title('Data on the Web', [4, 1, 1], 3).
unlabeled('A', [1, 5, 1, 1], 4).
em('fine', [2, 5, 1, 1], 4).
unlabeled('book.', [3, 5, 1, 1], 4).
year('2002', [2, 1], 3).
author('Buneman', [1, 2, 1], 3).
title('XML in Scotland', [2, 2, 1], 3).
unlabeled('The', [1, 1, 3, 2, 1], 6).
em('best', [2, 1, 3, 2, 1], 6).
unlabeled('ever!', [3, 1, 3, 2, 1], 6).

Here we can see the translation of each tag into a predicate name: *books*, *book*, etc. Each predicate has three 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. For instance, the three facts for the authors of the first book are numbered $[1, 1, 1]$, $[2, 1, 1]$ and $[3, 1, 1]$, representing the authors *'Abiteboul'*, *'Buneman'* and *'Suciu'*, respectively, and $[1, 2, 1]$ for representing *'Buneman'* in the second book (see Figure 1). Let us remark that the numbering in the facts is in reverse order with respect to the numbering in the node numbered XML document due to the use of lists for representing them.

The third argument of the predicate is a number used for numbering each type. The type number is needed to distinguish weakly distinct elements. For instance, the tag *review* has two rules, one for the case: "*A fine book*." and other one for the case " * The best ever*! **," where in the first case the sole emphasized text is *'fine'*, and in the second case all is emphasized, and *'best'* is doubled emphasized. The facts and rules in this case are:

---

*unlabeled('A', [1, 5, 1, 1], **4**).*
*em('fine', [2, 5, 1, 1], **4**).*
*unlabeled('book.', [3, 5, 1, 1], **4**).*
*unlabeled('The', [1, 1, 3, 2, 1], **6**).*
*em('best', [2, 1, 3, 2, 1], **6**).*
*unlabeled('ever!', [3, 1, 3, 2, 1], **6**).*

---

*review(reviewtype(Unlabeled,Em,[ ]),NodeReview, **3**):-*
        *unlabeled(Unlabeled,[NodeUnlabeled|NodeReview],**4**),*
        *em(Em,[NodeEm|NodeReview],**4**).*
*review(reviewtype(Em,[ ]),NodeReview,**3**):-*
        *em(Em,[NodeEm|NodeReview],**5**).*
*em(emtype(Unlabeled,Em,[ ]), NodeEms,**5**) :-*
        *unlabeled(Unlabeled,[NodeUnlabeled|NodeEms],**6**),*
        *em(Em, [NodeEm|NodeEms],**6**).*

They allow us to distinguish that the first case is built from the first *review* rule and the second from the second *review* rule–together with the *em* rule. Obviously, in highly nonstructured documents there could have many schema rules. The same happens in the case of the following XML document:

```
<books>
<book year=“2003”>
<author>Abiteboul</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>
</book>
</books>
```

where we have two kinds of records, one with *author*, *title*, *review*, and *year*, and the second one with *author*, *title*, and *year*. In this case, we have to consider the following schema rules:

```
books(bookstype(Book, [ ]), NodeBooks,1):-
        book(Book, [NodeBook|NodeBooks],2).
book(booktype(Author, Title, Review, [Year]), NodeBook,2) :-
        author(Author, [NodeAuthor|NodeBook],3),
        title(Title, [NodeTitle|NodeBook],3),
        review(Review, [NodeReview|NodeBook],3),
        year(Year, NodeBook,3).
book(booktype(Author, Title, [Year]), NodeBook,2) :-
        author(Author, [NodeAuthor|NodeBook],4),
        title(Title, [NodeTitle|NodeBook],4),
        year(Year, NodeBook,4).
author('Abiteboul',[1,1,1],3).
author('Buneman',[1,2,1],4).
...
```

The use of numbers **2-3-3-3-3** and **2-4-4-4** in the above rules, and in the corresponding facts, allows the distinction of the subelements of *Abiteboul* and *Buneman*'s books. The use of the same type numbering would suppose ambiguity, given that the *Abiteboul*'s book has also the type described by second rule of *book*.

On the other hand, whenever in a tagged element there is more than one value for the same subtag, we introduce one fact for each value, numbered with the same type number, but distinct node number. For instance, with respect to the running example:

```
author('Abiteboul', [1, 1, 1], 3).
author('Buneman', [2, 1, 1], 3).
author('Suciu', [3, 1, 1], 3).
```

In addition, the attributes of tagged elements are stored in a *Prolog* list. For instance, with respect to the following XML document:

<book year="2003",keyword="XML">
<author>Abiteboul</author>
<title>Data on the Web</title>
<review>A fine book.</review>
</book>

we will consider the following schema rule:

book(booktype(Author, Title, Review, **[Year,Keyword]** *), NodeBook, 2) :-*
   author(Author, [NodeAuthor|NodeBook],3),
   title(Title, [NodeTitle|NodeBook],3),
   review(Review, [NodeReview|NodeBook],3),
   year(**Year***,NodeBook,3),*
   keyword(**Keyword***,NodeBook,3).*

Finally, each value in a non-terminal tagged element is translated into a fact called *unlabeled*. This is the case in the running example of $'A'$ and $'book.'$ in the first review, and $'The'$ and $'ever!'$ in the second one.

4 Program Specialization for XPath Expressions

In this section, we will present the program specialization technique for querying *XPath* expressions against an XML document represented by means of a logic program. Firstly, we present the semantic of the *XPath* expressions.

4.1 XPath Semantics

An *XPath expression xpathexpr* has the form $/expr_1/\ldots/expr_n$ where each *simple XPath expression $expr_i$* has the form:

1. $expr \equiv tag$
2. $expr \equiv tag[cond]$
3. $expr \equiv @att$
4. $expr \equiv text()$

and *cond* is a boolean condition which has the form:

(a) $cond \equiv tag = value$
(b) $cond \equiv @att = value$
(c) $cond \equiv cond_1$ *and* $cond_2$
(d) $cond \equiv cond_1$ *or* $cond_2$
(e) $cond \equiv xpathexpr$

The above expressions $expr_i$ when $1 \leqslant i < n$ can only be chosen from the cases (1) and (2). We consider only a subset of *XPath* w.r.t. the *XPath* specification (W3C 2007b) which can specify paths on XML trees and restricts boolean conditions to express equalities to values connected with *"and"* and *"or"* logic connectives. This restriction is enough to understand our proposed technique. More complex *XPath*

queries can be translated into logic programming following similar ideas. We have implemented in our prototype a rich set of *XPath* queries including primitives "*", "//", "/../", ">","<", etc.

The semantics of the previous *XPath* expressions is as follows. Given an XML document, an *XPath* expression defines a subtree of the XML document. It can be defined as the subtree obtained from the XML tree satisfying each simple expression *expr* in the *XPath* expression. The semantics of *XPath* expressions could be defined as a forest (i.e. a sequence of subtrees) instead of a tree. However, we have adopted this definition in which an *XPath* expression defines a rooted document. The root is the same as the input document and therefore describes a *complete branch of the input document*. More concretely:

Given an XML document $\mathcal{X}$ and an *XPath* expression $xpathexpr = /expr_r \ldots /expr_n$ the *subtree of $\mathcal{X}$ defined by xpathexpr* is denoted by $subtree(\mathcal{X}, xpathexpr)$ and defined as:

(a) If $\mathcal{X}$ is a non terminal tagged element and has the form

$$< tag\ att_1 = v_1,\ldots,att_n = v_n > elem_1,\ldots,elem_s < /tag >$$

   then
   (a.1):

$$subtree(\mathcal{X}, /expr_r/\ldots/expr_n) =_{def}$$
$$< tag\ att_1 = v_1,\ldots,att_n = v_n >$$
$$subtree(elem_1, /expr_{r+1}/\ldots/expr_n),$$
$$\ldots,$$
$$subtree(elem_s, /expr_{r+1}/\ldots/expr_n),$$
$$elem_{i_1},$$
$$\ldots,$$
$$elem_{i_k}$$
$$< /tag >$$

   whenever $r < n$ and $\mathcal{X}$ satisfies $expr_r$; where $elem_{i_1},\ldots,elem_{i_k}$ is the subsequence of $elem_1,\ldots,elem_s$ satisfying *cond* whenever $expr_r \equiv tag[cond]$;
   (a.2):

$$subtree(\mathcal{X}, /expr_n) =_{def} \mathcal{X}$$

   whenever $r = n$ and $\mathcal{X}$ satisfies $expr_n$; and
   (a.3):

$$subtree(\mathcal{X}, /expr_r/\ldots/expr_n) =_{def} \epsilon$$

   otherwise.
(b) If $\mathcal{X}$ is a terminal tagged element then
   (b.1):

$$subtree(\mathcal{X}, /expr_r/\ldots/expr_n) =_{def} \mathcal{X}$$

   whenever $r = n$ and $\mathcal{X}$ satisfies $expr_r$; and
   (b.2):

$$subtree(\mathcal{X}, /expr_r/\ldots/expr_n) =_{def} \epsilon$$

   otherwise.

(c) If $\mathcal{X}$ has a basic type then
   (c.1):

$$subtree(\mathcal{X}, /text()) =_{def} \mathcal{X}$$

   and
   (c.2):

$$subtree(\mathcal{X}, xpathexpr) =_{def} \epsilon$$

   whenever $xpathexpr \not\equiv /text()$

where ϵ denotes the empty sequence.

In addition, an XML document $\mathcal{X}$ *satisfies a simple XPath expression expr* in the following cases:

(i) $\mathcal{X} \equiv < tag \ att_1 = v_1, \ldots, att_n = v_n > elem_1, \ldots, elem_s < /tag >$

satisfies *expr* whenever:

 (i.1) $expr \equiv tag$
 (i.2) $expr \equiv tag[cond]$ and $\mathcal{X}$ satisfies the condition *cond*, that is:
    (i.2.1) $cond \equiv tag' = value$ and tag' is a terminal tagged subelement of *tag* and the value of tag' is equal to *value*.
    (i.2.2) $cond \equiv @att = value$, some att_i $1 \leqslant i \leqslant n$ is equal to *att*, and v_i is equal to *value*.
    (i.2.3) $cond \equiv cond_1 \ and \ cond_2$, $\mathcal{X}$ satisfies the condition $cond_1$ and $\mathcal{X}$ satisfies the condition $cond_2$.
    (i.2.4) $cond \equiv cond_1 \ or \ cond_2$, $\mathcal{X}$ satisfies the condition $cond_1$ or $\mathcal{X}$ satisfies the condition $cond_2$.
    (i.2.5) $cond \equiv xpathexpr$ and $subtree(\mathcal{X}, /tag/xpathexpr)$ is a branch of $\mathcal{X}$.
 (i.3) $expr \equiv @att$ and some att_i $1 \leqslant i \leqslant n$ is equal to *att*

and

(ii) $\mathcal{X}$ *has a basic type*

satisfies *expr* whenever $expr \equiv text()$.

For instance, w.r.t. the running example, the *XPath* expression $/books/book[author = \text{“Suciu”}]/title$ defines $subtree(\mathcal{X}, /books/book[author = \text{“Suciu”}]/title)$ which is equal to:

$<books >$
$subtree(\mathcal{X}', /book[author=\text{“Suciu”}]/title)$
$subtree(\mathcal{X}'', /book[author=\text{“Suciu”}]/title)$
$</books>$

---

by case (a.1) of the definition, since there is no boolean conditions in *books*, where $\mathcal{X}'$ is:

<book year="2003">
<author>Abiteboul</author>
<author>Buneman</author>
<author>Suciu</author>
<title>Data on the Web</title>
<review>A fine book.</review>
</book>

and $\mathcal{X}''$ is:

<book year="2002">
<author>Buneman</author>
<title>XML in Scotland</title>
<review>The best ever!</review>
</book>

In addition, *subtree*($\mathcal{X}'$, /*book*[*author* = "*Suciu*"]/*title*) is equal to:

<book year="2003">
<author>Suciu</author>
subtree($\mathcal{X}'''$,/title)
</book>

by case (a.1) of the definition, given that the boolean condition [*author* = "*Suciu*"] is satisfied by < *author* > *Suciu* < /*author* >, by case (i.2.1) of definition, and is not satisfied by < *author* > *Abiteboul* < /*author* > and < *author* > *Buneman* < /*author* >. In addition, $\mathcal{X}'''$ is:

<title>XML in Scotland</title>

and *subtree*($\mathcal{X}''$, /*book*[*author* = "*Suciu*"]/*title*) = ϵ, by case (a.3) of the definition. Finally, *subtree*($\mathcal{X}'''$, /*title*) is equal to:

<title>XML in Scotland</title>

by case (a.2) of the definition. Therefore *subtree*($\mathcal{X}$, /*books*/*book*[*author* = "*Suciu*"]/ *title*) is equal to:

<books>
<book year="2003">
<author>Suciu</author>
<title>XML in Scotland</title>
</book>
</books>

In other words, the subtree defined by an *XPath* expression can be seen as the subtree of the input XML document which is traversed for answering the query. In practice, the answer to an *XPath* query consists of the sequence of subtrees (i.e. the forest) of the tree defined by the *XPath* expression, whose tag is equal to the rightmost tag of the *XPath* query. For instance, in the above example, the answer would be:

<center><i><title>XML in Scotland</title></i></center>

---

given that the rightmost tag of the *XPath* query is *title*.

4.2 Schema Rule Specialization

The *first step of the program specialization* consists of a predicate removing from the schema rules.

With this aim, we need to map each *XPath* expression to a so-called *free of equalities XPath expression*. Each *XPath* expression $xpathexpr = /expr_1 \ldots /expr_n$ can be mapped into a *free of equalities XPath expression* as follows.

Each simple *XPath* expression *expr* can be mapped into a *free of equalities simple XPath expression* denoted by $FE(expr)$. Analogously, we need to define $FE(cond)$ which is a *free of equalities boolean condition* associated to a boolean condition *cond*. They are defined as follows, distinguishing cases in the form of *expr* and *cond*.

1. $expr \equiv tag$: $FE(expr) =_{def} expr$.
2. $expr \equiv tag[cond]$: $FE(expr) =_{def} tag[FE(cond)]$
3. $expr \equiv @att$: $FE(expr) =_{def} @att$
4. $expr \equiv text()$: $FE(expr) =_{def} text()$
5. $cond \equiv tag = value$: $FE(expr) =_{def} tag$
6. $cond \equiv @att = value$: $FE(expr) =_{def} @att$
7. $cond \equiv cond_1 \ and \ cond_2$: $FE(expr) =_{def} FE(cond_1) \ and \ FE(cond_2)$
8. $cond \equiv cond_1 \ or \ cond_2$: $FE(expr) =_{def} FE(cond_1) \ or \ FE(cond_2)$
9. $cond \equiv xpathexpr$: $FE(expr) =_{def} FE(xpathexpr)$

Now, given $xpathexpr = /expr_1/ \ldots /expr_n$ then $FE(xpathexpr) =_{def} /FE(\ expr_1)/ \ldots /FE(expr_n)$. Free of equalities *XPath* expressions *xpathfree* are expressions $/fexpr_1/ \ldots /fexpr_n$ where each $fexpr_i$, $1 \leqslant i \leqslant n$, has the form:

1. $fexpr \equiv tag$
2. $fexpr \equiv tag[cond]$
3. $fexpr \equiv @att$
4. $fexpr \equiv text()$

and *cond* is a free of equalities boolean condition which has the form:

(a) $cond \equiv cond_1 \ and \ cond_2$
(b) $cond \equiv cond_1 \ or \ cond_2$
(c) $cond \equiv xpathfree$

Free of equalities *XPath* expressions define a subtree of the XML document in which some subpaths of the XML document must exist due to occurrences of free of equalities boolean conditions.

For instance, in the running example, $FE(/books/book\,[author = "Suciu"]/title)$ $= /books/book\,[author]\,/title$, and the subtree of the (type and node numbered) XML document which corresponds with the *XPath* expression $/books/book\,[author]$ $/title$ is as follows:

<books nodenumber=1, typenumber=1>
<book year="2003", nodenumber=1.1, typenumber=2>
<author nodenumber=1.1.1 typenumber=3>Abiteboul</author>
<author nodenumber=1.1.2 typenumber=3>Buneman</author>
<author nodenumber=1.1.3 typenumber=3>Suciu</author>
<title nodenumber=1.1.4 typenumber=3>Data on the Web</title>
</book>
<book year="2002" nodenumber=1.2, typenumber=2>
<author nodenumber=1.2.1 typenumber=3>Buneman</author>
<title nodenumber=1.2.2 typenumber=3>XML in Scotland</title>
</book>
</books>

Let us remark that the boolean condition [*author*] forces to include each author in the subtree represented by the free of equalities *XPath* expression $/books/book$ [*author*]/ *title*.

Now, given a type and node numbered XML document $\mathcal{X}$ and an *XPath* expression *xpathexpr* then the *specialized program* $\mathscr{P}^{xpathexpr}$ *obtained from* $\mathscr{P}$ is defined as the schema rules for the subtree of $\mathcal{X}$ defined by *xpathfree*, where *xpathfree* is the free of equalities *XPath* expression obtained from *xpathexpr*, together with the facts of $\mathscr{P}$. In other words:

$$\mathscr{P}^{xpathexpr} =_{def} Rules(Prog(subtree(\mathcal{X}, FE(xpathexpr)))) \cup Facts(\mathscr{P})$$

For instance, with respect to the running example and $/books/book$ [*author* = "*Suciu*"]/*title*, $\mathscr{P}^{/books/book[author="Suciu"]/title}$ consists of the specialized schema rules:

---

books(bookstype(Books, [ ]), NodeBooks,1):-
book(Book, [NodeBook|NodeBooks],2).
book(booktype(Author,Title,Review,[Year]),NodeBook,2) :-
author(Author,[NodeAuthor|NodeBook],3),
title(Title,[NodeTitle|NodeBook],3).

---

together with the set of facts of $\mathscr{P}$.

Let us remark that in practice, the specialized schema rules can be obtained from the schema rules by removing predicates; that is, removing the predicates in the schema rules which are not tags in the (free of equalities) *XPath* expression.

4.3 Generation of Goals

The *second step of the specialization program* consists of (1) to consider the equalities removed from the original *XPath* expression when the free of equalities *XPath* expression was generated, and (2) to generate a set of goals from these equalities.

With this aim, each *XPath* expression *xpathexpr* can be mapped into a set of *Prolog* terms, denoted by $PT(xpathexpr)$, denoting the set of *patterns of the query*. These patterns are instances of the *"elementname+type"* patterns defined in our translation.

In particular, each simple *XPath* expression *expr* can be mapped into a set of patterns, denoted by $PT(expr)$. This set can be defined as follows, distinguishing cases in the form of *expr*:

1. $expr \equiv tag$: $PT(expr) =_{def} \emptyset$.
2. $expr \equiv tag[cond]$:
   (a) $cond \equiv tag_i = value$: $PT(expr) =_{def} \{tagtype(\overline{Tag}, [\overline{Att}])\{Tag_i \to value\} \mid tagtype(\overline{Tag}, [\overline{Att}]) \in PT(tag)\}$.
   (b) $cond \equiv @att_i = value$: $PT(expr) =_{def} \{tagtype(\overline{Tag}, [\overline{Att}])\{Att_i \to value\} \mid tagtype(\overline{Tag}, [\overline{Att}]) \in PT(tag)\}$.
   (c) $cond \equiv cond_1$ *and* $cond_2$. $PT(expr) =_{def} \{t\theta \mid \theta = m.g.u.(t, t'), t \in PT(tag[cond_1]), t' \in PT(tag[cond_2]))\}$
   (d) $cond \equiv cond_1$ *or* $cond_2$. $PT(expr) =_{def} PT(tag [cond_1]) \cup PT(tag[cond_2])$
   (e) $cond \equiv xpathexpr$: $PT(expr) =_{def} PT(xpathexpr)$
3. $expr \equiv @att$: $PT(expr) =_{def} \emptyset$
4. $expr \equiv text()$: $PT(expr) =_{def} \emptyset$

Now,

$$PT(/expr_1/ \ldots /expr_n) =_{def} \{t_1\theta \mid \theta = m.g.u.(t_1, \ldots, t_n), t_i \in PT(expr_i), 1 \leqslant i \leqslant n\}$$

Now, given a type and node numbered XML document and an *XPath* expression *xpathexpr* then the *set of specialized goals for xpathexpr* is defined as the set:

$$\mathcal{G}^{xpathexpr} =_{def}$$

$$\{tag(Pattern, Node, Type)\{Pattern \to t, Type \to r\} \mid$$

$$t \in PT(xpathexpr), r \in TN(t)\}$$

where *tag* is the leftmost tag in *xpathexpr* with a boolean condition. If there is no boolean conditions, the set is defined as:

$$\mathcal{G}^{xpathexpr} =_{def}$$

$$\{tag(Pattern, Node, Type)\{Type \to r\} \mid$$

$$t \in PT(tag), r \in TN(t)\}$$

For instance, with respect to $/books/book\,[author = \text{``Suciu''}]/title$ and the running example $PT(/books/\,book[author = \text{``Suciu''}]/title) = \{booktype('Suciu', Title, Review, [Year])\}$ and $TN(booktype('Suciu', Title, Review, [Year])) = \{2\}$. Therefore the (unique) goal is : $-book(booktype('Suciu', Title, Review, [Year]), Node, 2)$.

In summary, the handling of an *XPath* query involves the specialization of the schema rules of the XML document and the generation of one or more goals. The goals are obtained from the leftmost tag in the *XPath* expression with a boolean condition, instantiated by mean of patterns obtained from the boolean equalities.

4.4 Reconstruction of the answer

In order to rebuild the answer, we have to reason as follows.

A logic program $\mathscr{P}$ obtained from an XML document $\mathscr{X}$ contains schema rules and facts of the form $att(value, i, r)$ and $tag(value, i, r)$, and conversely, from this set of facts and the schema rules we can rebuild the document $\mathscr{X}$.

However, the same (and fragments of the) XML document $\mathscr{X}$ can be also obtained from the schema rules and facts of the form $att(value, i, r)$ and $tag(t, i, r)$ whenever t's are *Prolog* terms of the form $tagtype(s, j, k)$, –t's are pattern instances– and $tag(t, i, r)$ belongs to the Herbrand model (with variables) of $\mathscr{P}$.

For instance, from the following fact:

book(booktype('Abiteboul', Title, reviewtype('A ', fine, [ ]),['2003']),[1,1],2).

and the schema rules of the running example, we can rebuild the XML document:

```
<books nodenumber=1, typenumber=1>
<book year="2003", nodenumber=1.1, typenumber=2>
<author nodenumber=1.1.1 typenumber=3>Abiteboul</author>
<review nodenumber=1.1.5 typenumber=3>
<unlabeled nodenumber=1.1.5.1 typenumber=4> A </ unlabeled>
<em nodenumber=1.1.5.2 typenumber=4>fine</em>
</review>
</book>
</books>
```

Let us remark that the previous fact represents a fragment of the whole XML document, where the type and node numbering together with the schema rules allow us to rebuild this fragment of the XML document. In this fact the variable *Title* represents a missing value in the XML document.

Therefore when a goal obtained from an *XPath* expression is called, each answer of the goal represents a fragment of the *XPath* query answer.

Given a type and node numbered XML document $\mathscr{X}$, the logic program $\mathscr{P}$ representing $\mathscr{X}$, and an *XPath* expression *xpathexpr*, then we can build the *XML document representing the answer*, denoted by $Doc(xpathexpr, \mathscr{P})$, as follows:

$$Doc(xpathexpr, \mathscr{P}) =_{def} Doc(Rules(\mathscr{P}^{xpathexpr}) \cup$$
$$\{tag(t, Node, r)\theta | \theta \text{ is an answer of } tag(t, Node, r),$$
$$\text{w.r.t. } \mathscr{P}^{xpathexpr}, tag(t, Node, r) \in \mathscr{G}^{xpathexpr} \})$$

Analogously, when the *XPath* expression *xpathexpr* has no boolean conditions:

$$Doc(xpathexpr, \mathscr{P}) =_{def} Doc(Rules(\mathscr{P}^{xpathexpr}) \cup$$
$$\{tag(X, Node, r)\theta | \theta \text{ is an answer of } tag(X, Node, r),$$
$$\text{w.r.t. } \mathscr{P}^{xpathexpr}, tag(X, Node, r) \in \mathscr{G}^{xpathexpr} \})$$

Let remark us that our programs have finite answers and thus the previous definition has sense. In addition, the previous definition defines the XML document answer of an *XPath* expression as a *complete branch of the input XML document*.

For instance, w.r.t. the running example and the *XPath* expression */books/book* [*author* = "*Suciu*"]/*title*, the (unique) goal is : −*book*(*booktype*(′*Suciu*′, *Title*, *Review*, [*Year*], *Node*, 2), and the (unique) answer of the goal w.r.t. the following specialized schema rule:

$$book(booktype(Author,Title,Review,[Year]),NodeBook,2) :-$$
$$author(Author,[NodeAuthor|NodeBook],3),$$
$$title(Title,[NodeTitle|NodeBook],3).$$

is $\theta = \{ Title / {'Data\ on\ the\ Web'}, Node / [1,1] \}$. Now, from the goal instance *book*(*booktype*(′*Suciu*′, ′*Data on the Web*′, *Review*, [*Year*], [1,1], 2) obtained from θ, we can rebuild the answer:

```
<books nodenumber=1, typenumber=1>
<book nodenumber=1.1, typenumber=2>
<author nodenumber=1.1.1 typenumber=3>Suciu</author>
<title nodenumber=1.1.4 typenumber=3>Data on the Web</title>
</book>
</books>
```

Therefore, the XML document representing the answer of an *XPath* expression is defined as the document obtained from the specialized schema rules and the goal instances obtained from each answer of the goals.

4.5 Reordering

Finally, there is an optimization in our proposed technique which consists in the re-ordering of predicates in the schema rules in order to follow a *left-to-right evaluation order* of *XPath* expressions. The aim of such left-to-right evaluation order is to keep the order of filtering that the user specifies by means of the boolean conditions.

For instance, in the case of the *XPath* expression */books/book*[@*year* = *2002* *and title* = "*Data on the Web*"]/*author*, the user has required the authors of the books published in the year *2002* with title *"Data on the Web."* Following a left-to-right evaluation order, firstly, the books are filtered by the year, and after by the title.

This predicate reordering is as follows. Supposing the *XPath* expression */books* /*book* [@*year* = *2002* *and title* = "*Data on the Web*"]/*author*, the schema rule specialization should correspond with:

$$book(booktype(Author, Title, Review, [Year]),NodeBook,2):-$$
$$author(Author, [NodeAuthor|NodeBook],3),$$
$$title(Title,[NodeTitle|NodeBook],3),$$
$$year(Year,NodeBook,3).$$

However, in order to follow a left-to-right evaluation order of the *XPath* expression, we reorder the predicates in the body of the predicate *book* and we transform this schema rule into:

---

book(booktype(Author, Title, Review, [Year]),NodeBook,2):-
year(Year,NodeBook,3),
title(Title,[NodeTitle|NodeBook],3),
author(Author, [NodeAuthor|NodeBook],3).

---

in which, firstly, the books are filtered by year, after the titles are obtained, and finally, the authors are computed.

4.6 Examples

In this section we would like to show some examples of the proposed technique. In each example, we will show the specialized schema rules, the set of generated goals, the set of answers, and the answer in the form of an XML document obtained from the goal instances.

Example 1

For instance, we can suppose an *XPath* query such as */books/book/author*, requiring the authors in the book database. In this case, we have to consider the unique goal $: -author(Author, Node, 3)$, given that $PT(author) = \{authortype(Author, [])\}$ and $TN(authortype(Author, [])) = \{3\}$. The call of such a goal will compute the answers:

---

(1) Author/'Abiteboul'	*Node/[1,1,1]*
(2) Author/'Buneman'	*Node/[2,1,1]*
(3) Author/'Suciu'	*Node/[3,1,1]*
(4) Author/'Buneman'	*Node/[1,2,1]*

---

which correspond with the following set of goal instances and XML document:

	<result>
author('Abiteboul', [1, 1, 1],3).	*<author>Abiteboul</author>*
author('Buneman', [2, 1, 1],3).	*<author>Buneman</author>*
author('Suciu', [3, 1, 1],3).	*<author>Suciu</author>*
author('Buneman', [1, 2, 1],3).	*<author>Buneman</author>*
	</result>

Let us remark that answer is packed into a tag called *result*.

Example 2

Now, we can suppose the *XPath* expression */books/book*. Now, the unique goal is: $-book(Book, Node, 2)$, because $PT(book) = \{booktype(Author, Title, Review, [Year])\}$ and $TN(booktype(Author, Title, Review, [Year])) = \{2\}$. The call of the goal *book(Book, Node,2)* computes the following answers:

(1) *Book/booktype('Abiteboul', 'Data on the Web', reviewtype('A', 'fine', [ ]),['2003'])*
Node/[1, 1]
(2) *Book/booktype('Abiteboul', 'Data on the Web', reviewtype('book.', 'fine', []), ['2003'])*
Node/[1, 1]
(3) *Book/booktype('Buneman', 'Data on the Web', reviewtype('A', 'fine', [ ]), ['2003'])*
Node/[1, 1]
(4) *Book/booktype('Buneman', 'Data on the Web', reviewtype('book.', 'fine', [ ]), ['2003'])*
Node/[1, 1]
(5) *Book/booktype('Suciu', 'Data on the Web', reviewtype('A', 'fine', [ ]), ['2003'])*
Node/[1, 1]
(6) *Book/booktype('Suciu', 'Data on the Web', reviewtype('book.', 'fine', [ ]), ['2003'])*
Node/[1, 1]
(7) *Book/booktype('Buneman', 'XML in Scotland', reviewtype(emtype('The', 'best', [ ]), [ ]), ['2002'])*
Node/[2, 1]
(8) *Book/booktype('Buneman', 'XML in Scotland', reviewtype(emtype('ever!', 'best', [ ]), [ ]), ['2002'])*
Node/[2, 1]

which corresponds with the following document:

```
<result>
<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>
</result>
```

Example 3

Let us consider the *XPath* expression */books/book* [*author* = "*Suciu*"]/*title*. In this case, we have a condition in the form of *author* = "*Suciu*".

Therefore we have to consider (a) the goal : $-book(booktype('Suciu', Title, Review, [Year]), Node, 2)$ given that $PT(/books/book$ [*author* = "*Suciu*"]/*title*$) = \{booktype('Suciu', Title, Review, [Year])\}$ and $TN(booktype('Suciu', Title, Review, [Year])) = \{2\}$; and we have to consider (b) the following specialized rule:

book(booktype(Author,Title,Review,[Year]),NodeBook,2) :-
author(Author,[NodeAuthor|NodeBook],3),
title(Title,[NodeTitle|NodeBook],3).

In the evaluation, the goal will firstly trigger the retrieval of the books for the author *'Suciu.'* In particular, it will retrieve the node numbers of *Suciu*'s books. It is achieved due to the instantiation of the corresponding argument in the goal.

Afterward, it allows us the retrieval of *Suciu′s* book titles, ensuring that Suciu's book titles are the only computed ones.

The use of *author*(*Author*, [*NodeAuthor*|*NodeBook*], 3) is vital for the efficient retrieval of such titles, given that the node number has been instantiated in this predicate in the first step. In this case, the first used fact is *author*(′*Suciu*,′ [3, 1, 1], 3) with the node number [3, 1, 1] and this node number is used for retrieving the fact *title*(′*Data on the Web*,′ [4, 1, 1], 3). Next, we show the (unique) computed answer by means of the evaluation as well as the XML document represented by the goal instance:

Title/′*Data on the Web*′ *Review*/*Review*′ *Year*/*Year*′ *Node*/[1, 1]	*<result>* *<title>Data on the Web</title>* *</result>*

Let us remark that in the position of *year* and *review*, which are not required in the *XPath* expression, the goal returns variables (i.e. *Review′*, *Year′*). That is, the evaluation does not use the facts for these elements. This is the main effect of our specialization technique.

Example 4

Now, let us consider the *XPath* query /*books*/*book* [@*year* = *2002 and title* = "*Data on the Web*"]/*author*. In this case, the goal is : −*book* (*booktype*(*Author*,′ *Data on the Web′*, *Review*, [′*2002′*]), *Node*, 2), and the specialized schema rule is:

$$book(booktype(Author, Title, Review, [Year]), NodeBook, 2) :-$$
$$year(Year, NodeBook, 3),$$
$$title(Title, [NodeTitle|NodeBook], 3),$$
$$author(Author, [NodeAuthor|NodeBook], 3).$$

In this specialized schema rule, we can see that the call to *review* has been removed from the original schema rule, and the predicates have been reordered with the aim of following the same order as the *XPath* expression. That is, the boolean conditions are checked from left to right (firstly, @*year* = *2002* and after *title* = "*Data on the Web*"), and finally, the authors are computed. In other words, starting from the goal *book*(*booktype* (*Author*,′ *Data on the Web′*, *Review*, [′*2002′*]), *Node*, 2), firstly the retrieval of the books for the year *2002* is triggered. Afterward, the retrieval of titles for this year (using the node number instantiated in the previous step) is triggered; concretely the book titled "*Data on the Web*". Finally, the authors of such books are retrieved using node numbers instantiated in the previous steps.

In the case of an "or" connective, that is, /*books*/*book* [@*year* = *2002 or title* = "*Data on the Web*"]/*author*, we would have two goals and patterns: : −*book* (*booktype* (*Author*,′ *Data on the Web′*, *Review*, [*Year*]), *Node*, 2) and : −*book* (*booktype* (*Author*, *Title*, *Review*, [′*2002′*]), *Node*, 2).

Example 5

Let us consider the *XPath* query /*books*/*book* [@*year* = *2002*]/*author* [*name* = "*Serge*"] with respect to the following XML document:

```
<books>
    <book year="2003">
        <author>Abiteboul<name>Serge</name></author>
        <title>Data on the Web</title>
        <review>A <em>fine</em> book.</review>
    </book>
    <book year="2002">
        <author>Buneman <name>Peter</name></author>
        <title>XML in Scotland</title>
    </book>
</books>
```

In this case, we have two goals: : $-book(booktype\ (authortype(Unlabeled, 'Serge', []),$ $Title,\ Review,\ ['2002']),\ Node, 2)$ and : $-book(booktype\ (authortype(Unlabeled, 'Serge', []),\ Title,\ ['2002']),\ Node, 3)$. There are two goals because there are two weakly distinct records for the tag $book$: the first one has the subelement $review$ but not the second one.

In this case, there are two patterns for the query, that is, $PT(/books/book[@year = 2002]/author\ [name = "Serge"]) = \{booktype\ (authortype(Unlabeled, 'Serge', []),$ $Title,\ Review, ['2002']), booktype\ (authortype(Unlabeled, 'Serge', []),\ Title, ['2002'])\}$. In addition, there are two type numbers, one for each pattern $TN(booktype$ $(authortype(Unlabeled, 'Serge', []),\ Title,\ Review, ['2002'])) = \{2\}$ and $TN(booktype$ $(authortype(Unlabeled, 'Serge', []),\ Title,\ ['2002'])) = \{3\}$. Now, the specialized schema rules are:

```
book(booktype(Author, Title, Review, [Year]),NodeBook,2):-
        author(Author, [NodeAuthor|NodeBook],3),
        year(Year,NodeBook,3).
book(booktype(Author, Title, [Year]),NodeBook,3):-
        author(Author, [NodeAuthor|NodeBook],4),
        year(Year,NodeBook,4).
author(authortype(Unlabeled,Name,[ ]),NodeAuthor,3):-
        name(Name,[NodeName|NodeAuthor],4),
        unlabeled(Unlabeled,[NodeUnlabeled|NodeAuthor],4).
author(authortype(Unlabaled,Name,[ ]),NodeAuthor,4):-
        name(Name,[NodeName|NodeAuthor],5),
        unlabeled(Unlabeled,[NodeUnlabeled|NodeAuthor],5).
```

5 Theoretical Results

In this section, we will prove the correctness of the proposed technique. Our technique is correct in the sense that given a type and node numbered XML document $\mathcal{X}$, the logic program $\mathcal{P}$ represented by $\mathcal{X}$, and an $XPath$ expression $xpathexpr$ then $subtree(\mathcal{X}, xpathexpr) = Doc(xpathexpr, \mathcal{P})$. In other words, the subtree of an XML document defined by means of an $XPath$ expression is the same as the fragment of XML document build from the answers (w.r.t. the specialized schema rules) of the set of goal instances obtained from the same $XPath$ expression.

Theorem 1 (*Correctness*)

Given a type and node numbered XML document $\mathcal{X}$, the logic program $\mathcal{P}$ represented by $\mathcal{X}$, and an *XPath* expression *xpathexpr*, then $subtree(\mathcal{X}, xpathexpr) = Doc(xpathexpr, \mathcal{P})$.

Proof

Let *xpathexpr* be the *XPath* expression and let *xpathfree* = $FE(xpathexpr)$ be the free of equalities *XPath* expression associated to *xpathexpr*. Now, we have (1):

$$Doc(xpathexpr, \mathcal{P}) =$$

$$Doc(Rules(\mathcal{P}^{xpathexpr}) \cup \{tag(t, Node, r)\theta | tag(t, Node, r) \in \mathcal{G}^{xpathexpr}\})$$

by definition, where the θ's are answers w.r.t. $\mathcal{P}^{xpathexpr}$ and t is a variable whenever *xpathexpr* has no boolean conditions. Moreover, (2):

$$\mathcal{P}^{xpathexpr} = Rules(Prog(subtree(\mathcal{X}, xpathfree))) \cup Facts(\mathcal{P})$$

by definition. Let $\mathcal{F}$ be the set of facts used in the answers θ of $tag(t, Node, r)$:

$$\mathcal{F} =_{def} \{f\theta | f \in Facts(\mathcal{P}), f \text{ is a subgoal of } tag(t, Node, r) \text{ in the branch of } \theta,$$

$$tag(t, Node, r) \in \mathcal{G}^{xpathexpr}\}$$

Therefore, from (1) and (2), we have (3):

$$Doc(xpathexpr, \mathcal{P}) = Doc(Rules(\mathcal{P}^{xpathexpr}) \cup \mathcal{F})$$

Now, we have to prove that (4):

$$Doc(Rules(\mathcal{P}^{xpathexpr}) \cup \mathcal{F}) = Doc(Rules(Prog(subtree(\mathcal{X}, xpathexpr)))$$

$$\cup Facts(Prog(subtree(\mathcal{X}, xpathexpr))))$$

To prove (4) we have to reason that (5):

$$\mathcal{X}' = < tag' \ att_1 = v_1, \ldots, att_n = v_n, nodenumber = i, typenumber = k >$$

$$elem_1, \ldots, elem_s < /tag >$$

is a non terminal tagged subelement in $subtree(\mathcal{X}, xpathexpr)$ iff the schema rule

$$tag'(tagtype'(\overline{Tag}, [\overline{Att}]), Node, k) : -C \in Rules(Prog(subtree(\mathcal{X}, xpathexpr)))$$

where C is built from the tags of $elem_1, \ldots, elem_s$ and $att_1, \ldots, att_n$; and $\mathcal{X}'$ satisfies $expr_r$ where $xpathexpr = /expr_1 \ldots /expr_r / \ldots /expr_m$; and, in addition, (6):

$$\mathcal{X}' = < tag' \ nodenumber = i, typenumber = k > elem < /tag >$$

is a terminal tagged element in $subtree(\mathcal{X}, xpathexpr)$ iff

$$tag'(elem, i, k) \in \mathcal{F}$$

(5) is obvious by definition. Let us prove (6). We have to reason that if f is a subgoal of $tag(t, Node, r)$ and θ is the answer of the branch including f as

subgoal, then if $f\theta$ is a fact we can map $f\theta$ into a terminal tagged subelement of $subtree(\mathcal{X}, xpathexpr)$. It follows from the specialization of the schema rules of $\mathscr{P}$ and the choice of the patterns for *tag*.

Now, from (5) and (6) we can conclude (4) because if $\mathcal{X}'$ satisfies $expr_r$ then $\mathcal{X}'$ satisfies $FE(expr_r)$ by the definition of satisfiability, and therefore also:

$$tag'(tagtype'(\overline{Tag}, [\overline{Att}]), Node, k) : -C \in Rules(Prog(subtree(\mathcal{X}, xpathfree)))$$

and by (1):

$$Rules(\mathscr{P}^{xpathexpr}) = Rules(Prog(subtree(\mathcal{X}, xpathfree)))$$

Now, from (3) and (4), and taking into account that:

$$subtree(\mathcal{X}, xpathexpr) = Doc(Rules(Prog(subtree(\mathcal{X}, xpathexpr))))$$

$$\cup Facts(Prog(subtree(\mathcal{X}, xpathexpr))))$$

which is trivially true, then we can conclude that:

$$subtree(\mathcal{X}, xpathexpr) = Doc(xpathexpr, \mathscr{P})$$

6 Indexing

In this section, we will describe how to index XML documents represented by means of a logic program. In addition, we will show how to combine indexing and top-down evaluation. The aim of the indexing is to improve the retrieval of facts from secondary memory and therefore the execution of *XPath* queries.

In summary, the storing model in our approach is as follows:

- We use *main memory* for the storing of schema rules.
- We use *secondary memory (i.e. files)* for the storing of facts.
- We *index* facts in secondary memory.
- We have *two kinds of indexes*: one for indexing *predicate names*, and other one for indexing *group of facts*.

The use of main memory for storing the schema rules is justified due to in most of cases the number of schema rules is small. The use of secondary memory for storing facts is justified since XML documents can be too big in order to be stored in main memory.

Fact indexing is justified for efficiency reasons. Firstly, our approach requires to recover facts for a given predicate; in this case we use the first kind of index. Secondly, our approach requires to retrieve the elements grouped in the same XML record (i.e. groups of facts refereed to the same XML record); in this case we use the second kind of index.

For instance, w.r.t. the running example, we generate the following set of indexes:

first index	second index	group identifier	facts
author	*pos(1, 0).* *pos(2, 0).* *pos(3, 0).* *pos(9, 8).*	*[1, 1]*	*(0) year('2003', [**1, 1**], 3).* *(1) author('Abiteboul', [1, **1, 1**], 3).* *(2) author('Buneman', [2, **1, 1**], 3).* *(3) author('Suciu', [3, **1, 1**], 3).* *(4) title('Data on the Web', [4, **1, 1**], 3).*
em	*pos(6,5).* *pos(12,11).*	*[5,1,1]*	*(5) unlabeled('A ', [1, **5, 1, 1**], 4).* *(6) em(fine, [2, **5, 1, 1**], 4).* *(7) unlabeled(' book.', [3, **5, 1, 1**], 4).*
title	*pos(4, 0).* *pos(10, 8).*	*[2,1]*	*(8) year('2002', [**2, 1**], 3).* *(9) author('Buneman', [1, **2, 1**], 3).* *(10) title('XML in Scotland', [2, **2, 1**], 3).*
unlabeled	*pos(5, 5).* *pos(7, 5).* *pos(11, 11).* *pos(13, 11).*	*[1,3,2,1]*	*(11) unlabeled('The ', [1, **1, 3, 2, 1**], 6).* *(12) em(best, [2, **1, 3, 2, 1**], 6).* *(13) unlabeled(' ever!', [3, **1, 3, 2, 1**], 6).*
year	*pos(0, 0).* *pos(8, 8).*		

The first index allows the retrieval of facts by means of the predicate name: *author*, *year*, and so on. Therefore, the *first index key* is the *name of the predicate* and the *first index value* is the *set of relative positions in the file* of the facts for the predicate.

The second index allows to recover the relative position in the file of the group in which a fact is included. Therefore the *second index key* is the *relative position of the fact in the file* and the *second index value* is the *relative position in the file of the group in which the fact is included*.

With this aim the first index stores for each predicate name annotations of the form $pos(n, m)$, in which n denotes the relative position in the file of a fact for the predicate and m the relative position in the file of the group of this fact (therefore the second index is a secondary index).

For instance, *author* facts are stored in positions *1*, *2*, *3* and *9*, given by the annotation $pos(\mathbf{1}, 0)$, $pos(\mathbf{2}, 0)$, $pos(\mathbf{3}, 0)$, $pos(\mathbf{9}, 8)$, and the group of each author, that is, the XML record in which the author is included, starts at positions *0*, *0*, *0* and *8*, respectively, given by the annotations $pos(1, \mathbf{0})$, $pos(2, \mathbf{0})$, $pos(3, \mathbf{0})$, $pos(9, \mathbf{8})$. Each "group of facts" shares the *node number of the record*, which can be considered as the *identifier of the group*.

For instance, w.r.t. the running example, the first group can be identified by $[1, 1]$, and contains facts numbered as $[1, 1]$, $[1, 1, 1]$, $[2, 1, 1]$, $[3, 1, 1]$ and $[4, 1, 1]$. The second group is $[5, 1, 1]$, and so on. The reason for this grouping criteria is that each group of facts will be retrieved by means of the same schema rule. For instance, in the running example, the schema rule:

---

book(booktype(Author, Title, Review, [Year]), NodeBook ,2) :-
author(Author, [NodeAuthor|NodeBook],3),
title(Title, [NodeTitle|NodeBook],3),
review(Review, [NodeReview|NodeBook],3),
year(Year, NodeBook,3).

---

will retrieve the groups of facts [1, 1] and [2, 1].

Now, we will explain how the indexing technique is combined with the top-down evaluation of the goals. For instance, let us suppose the following *XPath* query: */books/book* [@*year = 2002 and author = "Buneman"*]*/review* w.r.t. the running example. Now, the specialized schema rules and facts used in the evaluation are:

(a) book(booktype(Author, Title, Review, [Year]), NodeBook ,2) :-
year(Year, NodeBook,3),
author(Author, [NodeAuthor|NodeBook],3),
review(Review, [NodeReview|NodeBook],3).
(b) review(reviewtype(Unlabeled,Em,[ ]),NodeReview,3):-
unlabeled(Unlabeled,[NodeUnlabeled|NodeReview],4),
em(Em,[NodeEm|NodeReview],4).
(c) review(reviewtype(Em,[ ]),NodeReview,3):-
em(Em,[NodeEm|NodeReview],5).
(d) em(emtype(Unlabeled,Em,[ ]),NodeEms,5) :-
unlabeled(Unlabeled,[NodeUnlabeled|NodeEms],6),
em(Em, [NodeEm|NodeEms],6).

(0) year('2003', [1, 1], 3).
(1) author('Abiteboul', [1, 1, 1], 3).
(2) author('Buneman', [2,1, 1], 3).
(3) author('Suciu', [3,1,1], 3).
(4) title('Data on the Web', [4, 1, 1], 3).
(5) unlabeled('A', [1, 5, 1, 1], 4).
(6) em('fine', [2, 5, 1, 1], 4).
(7) unlabeled('book.', [3, 5, 1, 1], 4).
(8) year('2002', [2, 1], 3).
(9) author('Buneman', [1, 2, 1], 3).
(10) title('XML in Scotland', [2, 2, 1], 3).
(11) unlabeled('The', [1, 1, 3, 2, 1], 6).
(12) em('best', [2, 1, 3, 2, 1], 6).
(13) unlabeled('ever!', [3, 1, 3, 2, 1], 6).

The combination of indexing and top-down evaluation can be summarized as follows. In general, the evaluation will generate (sub)goals which have the form: $tag(\_, [Var_1, \ldots, Var_n, N_1, \ldots, N_m], \_)$, where *tag* is a tag of the XML document. The second argument of such (sub)goals is a list of the form $[Var_1, \ldots, Var_n, N_1, \ldots, N_m]$ representing a *partially instantiated node number*, in which $Var_1, \ldots, Var_n$ are variables and $N_1, \ldots, N_m$ are natural numbers. There is a particular case of goals of the form $tag(\_, Var, \_)$, in which there is a variable in the second argument instead of a list. This particular case corresponds with the main goal.

In addition, each time a fact is recovered, the system stores, together with the identifier of its group, the relative position in the file of its group. For instance,

w.r.t. the running example, whenever $author('Buneman', [2,1,1], 3)$ is recovered, the system stores that the group $[1,1]$ is at position 0 in the file.

Now, the index accessing can be summarized as follows. Each time a subgoal $tag(\_, [Var_1, \ldots, Var_n, N_1, \ldots, N_m], \_)$ is called and does not unify with an schema rule then:

(a) Whenever $[Var_2, \ldots, Var_n, N_1, \ldots, N_m]$ *matches to a previously stored group identifier*, the system uses the relative position of the matched group for the retrieval of facts for *tag*. Therefore the second index is used for the retrieval of the facts.

(b) Whenever the stored group identifiers *do not match* to $[Var_2, \ldots, Var_n, N_1, \ldots, N_m]$, the system uses the first index for the retrieval of the elements of *tag*.

In the case of the main goal $tag(\_, Var, \_)$, the first index will be ever used.

Now, we show the *trace of the execution* of the *XPath* query $/books/book[@year = 2002$ and author $=$ "Buneman"]$/review$ with respect to the above indexing structure.

1. call of $book(booktype(Buneman, \_G12073, \_G12074, [2002]), \_G12078, 2)$ (Rule a)
2. call of $year(2002, \_G12128, 3)$ (Rule a)
3. first index accessing to position 0 due to $year(2002, \_G12128, 3)$; recovering $year(2003, [1,1], 3)$; **fail**.
4. first index accessing to position 8 due to $year(2002, \_G12128, 3)$; recovering $year(2002, [2,1], 3)$; storing that the position of group $[2,1]$ is 8; **success**.
5. call of $author(\,Buneman, [\_G12100, 2, 1], 3)$ (Rule a)
6. second index accessing to position 8 due to the position of group $[2,1]$ is 8; recovering $author(\,Buneman, [1,2,1], 3)$; **success**
7. call of $review(\_G12151, [\_G12148, 2, 1], 3)$ (Rule a)
8. call of $unlabeled(\_G12190, [\_G12187, \_G12212, 2, 1], 4)$ (Rule b)
9. first index accessing to position 11 due to $unlabeled(\_G12243, [\_G12240, \_G12265, \_G12268, 2, 1], 6)$; recovering $unlabeled(The, [1,1,3,2,1], 6)$; storing that the position of group $[1,3,2,1]$ is 11; **success**.
10. first index accessing to position 13 due to $unlabeled(\_G12243, [\_G12240, \_G12265, \_G12268, 2, 1], 6)$; recovering $unlabeled(ever!, [3,1,3,2,1], 6)$; storing that position of group $[1,3,2,1]$ is 11; **success**
11. call of $em(\_G12261, [\_G12258, \_G12283, \_G12286, 2, 1], 6)$ (Rule c)
12. second index accessing to position 11 due to $em(\_G12261, [\_G12258, \_G12283, \_G12286, 2, 1], 6)$ and that position of group $[1,3,2,1]$ is 11; recovering $em(best, [2,1,3,2,1], 6)$; **success**
13. $em(emtype(The, best, []), [1,3,2,1], 5)$ **success**
14. $em(emtype(ever!, best, []), [1,3,2,1], 5)$ **success**
15. $review(reviewtype(emtype(The, best, []), []), [3,2,1], 3)$ **success**
16. $review(reviewtype(emtype(ever!, best, []), []), [3,2,1], 3)$ **success**
17. $book(booktype(Buneman, \_G12316, reviewtype(emtype(The, best, []), []), [2002]), [2,1], 2)$ **success**
18. $book(booktype(Buneman, \_G12316, reviewtype(emtype(ever!, best, []), []), [2002]), [2,1], 2)$ **success**

7 Prototype

Now, we will show our prototype, named *XIndalog*. This prototype implements the technique presented in this paper. In addition, we have implemented a rich set of *XPath* queries including *XPath* constructions like "//", "/../"."*", etc. The prototype has been developed under *SWI-Prolog* (Wielemaker 2005) and hosted in a Web site at http://indalog.ual.es/Xindalog. This Web site has been developed by using a *CGI* (*Common Gateway Interface*) application, in order to link the Web site with the prototype. From the main page of the prototype (see Figure 2), we can access to a basic description of *XIndalog*, *XML*, *XPath*, as well as the *demo*.

Fig. 2. `http://indalog.ual.es/Xindalog`.

Fig. 3. Top-Down demo.

We have implemented two releases of the prototype: a top-down and bottom-up release (details about the later can be found in (Almendros-Jiménez *et al.* 2006)). In the Web site, there are some built-in examples which can be tested and new examples can also be typed.
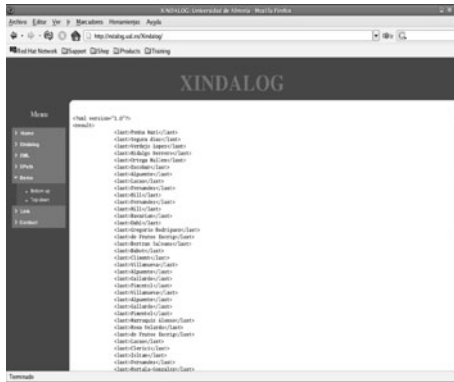

Fig. 4. Query example.

Fig. 5. Query result.

7.1 Benchmarks

We have tested our prototype by means of not enough structured XML documents and by means of XML documents of big size. Firstly, we have tested our prototype with a small but not enough structure XML document, shown in Table 1. Now and w.r.t. this document, we have considered the following set of *XPath* queries.

XPath Query		Meaning
⊙ /books/book[@year and @pages]/*		To obtain the books which have publishing year and number of pages
⊙ /books/book/author/@*		To obtain all the attributes of the authors
⊙ //book		To obtain all the books included in the *XML* document

XPath Query		Meaning
⊙ //book[review="Very good"]/author		To obtain all the authors of books with a very good review
⊙ //@year		To obtain all the years occurring in the XML document
⊙ /books/*/author		To obtain all the authors inside book records

XPath Query		Meaning
⊙ /books/book[review="Good"]/ author[name="John Durant"]		To obtain all the author information whose name is John Durant and the review is good
⊙ /books[book="The first book"]/book [@year=2003 and review="Good"] /author[name="Benz"]/../..		To obtain the books of the year 2003 and good review whose author is Benz
⊙ /books/book/text()		To obtain the books with textual information
⊙ /books/book[author/name]/title		To obtain the book titles whenever the books have author name
⊙ /books/(book | book2)/(review2 | review)		To obtain the reviews of the two kinds of books
⊙ /books/book/(author | title)		To obtain the book authors and titles
⊙ /books/(book | book2)//text()		To obtain the textual information from the two kinds of books
⊙ //@*		To obtain all the attributes of the document
⊙ /*/*/title		To obtain the titles that are at 3rd level
⊙ /*/*//*		To obtain all the elements and their nested from the 3rd level
⊙ /*/book2/*		To obtain all information from book2 at 2nd level
⊙ //*//author/..		To obtain the records containing author information from the 1st level

Table 1. *A small XML document*

```
<books year="2006">
    A book collection
    <book>empty</book>
    <book year="2003" pages="984">
        The first book
        <author english="yes" spanish="yes">
            Benz
            <name>Brian</name>
        </author>
        <author>John Durant</author>
        <author>John Durant</author>
        <title>XML Programming Bible</title>
        <review>Good</review>
    </book>
    <book year="2002">
        The second book
        <author>Dino Esposito</author>
        <title>Applied XML Programming for Microsoft .NET</title>
        <review>Good</review>
    </book>
    <book>
    The third book
        <author>Apt, Krzystof R.</author>
        <title>The Logic Programming Paradigm and Prolog</title>
        <review>Very good</review>
    </book>
    <book year="1994" pages="560">
        The fourth book
        <author english="yes" spanish="no">
            Leon Sterling
        </author>
        <author>Ehud Shapiro</author>
        <title>The Art of Prolog</title>
        <review>Very good</review>
    </book>
    <book2 year="2001">
        The fifth book
        <author english="yes">
            Elliotte Rusty Harold
        </author>
        <title>XML Bible</title>
        <review2>Good</review2>
    </book2>
    <book year="2003" pages="984">
        The first book
        <author english="yes" spanish="yes">
            Benz
            <name2>Brian</name2>
            <firstname>
            Brian
            <lastname>Benz</lastname>
            <others>no more</others>
            </firstname>
        </author>
        <author>John Durant</author>
        <author>John Durant</author>
        <title>XML Programming Bible</title>
        <review>Very good 2</review>
    </book>
</books>
```

Secondly, we have tested our prototype with XML documents of big size in order to get benchmarks, considering the following file sizes:

- *64KB*; 516 elements were included into the file;
- *128KB*; 1032 elements were included into the file;

- *256KB*; 2064 elements were included into the file;
- *512KB*; 4128 elements were included into the file; and finally,
- *1024KB*; 8256 elements.

For each file size, we have computed the following answer times:

- *Translation time*;
  It represents the time needed for translating a *XML* document into *Prolog* facts and rules;
- *Evaluation time*;
  It represents the time of the top-down evaluation of the (specialized) program w.r.t. an *XPath* query;
- *Browsing time*;
  It represents the time needed for formatting and browsing the query result.

Next, we will show three *XPath* queries with their corresponding times for each considered file size.

XPath Query: */books*

File size	Translation	Evaluation	Browsing	Total time
64KB	1,063sg	2,062sg	0,063sg	3,188sg
128KB	3,375sg	7,717sg	0,125sg	11,2171sg
256KB	11,860sg	31,296sg	0,312sg	43,4681sg
512KB	42,812sg	2min 11,110sg	0,578sg	2min 54,500sg

XPath Query: */books/book/title*

File size	Translation	Evaluation	Browsing	Total time
64KB	1,030sg	0,204sg	0,030sg	1,264sg
128KB	3,343sg	0,673sg	0,047sg	4,063sg
256KB	11,546sg	2,484sg	0,048sg	14,078sg
512KB	42,813sg	9,562sg	0,188sg	52,563sg

XPath Query: */books/book[review="very good"]/title*

File size	Translation	Evaluation	Browsing	Total time
64KB	1,046sg	0,032sg	0,0sg	1,078sg
128KB	3,359sg	0,063sg	0,0sg	3,422sg
256KB	11,579sg	0,108sg	0,0sg	11,687sg
512KB	42,796sg	0,188sg	0,0sg	42,984sg

The following tables show the benchmarks of the query */books/book[review="good"] /title* with and without our program specialization technique. From these tables,

we can conclude that *our specialization technique significantly improves the answer times.*

XPath Query: */books/book[review="good"]/title*

Without Program Specialization

File size	Translation	Evaluation	Browsing	Total time
64KB	0,750sg	1,562sg	0,046sg	2,358sg
128KB	2,095sg	5,202sg	0,095sg	7,392sg
256KB	6,579sg	19,407sg	0,187sg	26,173sg
512KB	22,530sg	1min 21,172sg	0,500sg	1min 44,202sg
1024KB	1min 22sg	5min 32,843sg	0,921sg	6min 55,764sg

With Program Specialization

File size	Translation	Evaluation	Browsing	Total time
64KB	0,750sg	0,172sg	0,015sg	0,937sg
128KB	2,079sg	0,546sg	0,0165sg	2,641sg
256KB	6,484sg	2sg	0,048sg	8,532sg
512KB	22,298sg	7,656sg	0,094sg	30,048sg
1024KB	1min 21,546sg	30,296sg	0,188sg	1min 52,030sg

8 Conclusions and Future Work

In this paper, we have presented how to represent and index XML documents by means of logic programming. Moreover, we have studied how to specialize a logic program, and how to generate goals in order to solve *XPath* queries. We have described how to use the indexing of the XML documents in order to obtain a more efficient top-down evaluation and query solving. Finally, we have shown benchmarks of our prototype developed with the proposed technique. Our approach opens two promising research lines:

- The first one, the extension of *XPath* to a more powerful query language such as *XQuery*, that is, the study of the implementation of *XQuery* in logic programming.

  We have developed an extension to *XQuery* in a recent paper (Almendros-Jiménez *et al.* 2007), which uses as basis the specialization technique studied here for *XPath* queries. *XQuery* enriches our proposal since in *XQuery* the queries can involve more than one XML document. In addition, *XQuery* allows us to express more complex queries w.r.t. a sole document. Now, we are developing the implementation of our new proposal.

- The second one, the use of logic *programming* as inference engine for the so-called *"Semantic Web"*, by introducing *RDF* documents or *OWL* specifications. In this line we are interested in the representation in logic *programming* of ontologies.

  There are some recent works (Wolz 2004; Grosof *et al.* 2003; Horrocks and Patel-Schneider 2004) interested in the identification of the intersection of logic programming and the so-called *Description Logic (DL)* (Borgida 1996), the basis of most ontology languages. The quoted proposals translate restricted forms of ontologies (i.e. restricted forms of *OWL* and therefore fragments of *DL*) into logic *programming*. Our work can be integrated in this framework by combining our logic *programming* based transformation of XML documents and the transformation of ontologies into logic *programming*.

  The interest of such integration is to provide semantic information about XML documents, the use of such semantic information in order to inferring new information, and thus to improve the answers to *XPath* and *XQuery* queries.

References

ABITEBOUL, S., BUNEMAN, P. AND SUCIU, D. 2000. *Data on the Web: From Relations to Semistructured Data and XML.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

ALMENDROS-JIMÉNEZ, J. M., BECERRA-TERÓN, A. AND ENCISO-BAÑOS, F. J. 2006. Magic sets for the XPath language. *Journal of Universal Computer Science 12,* 11, 1651–1678.

ALMENDROS-JIMÉNEZ, J. M., BECERRA-TERÓN, A. AND ENCISO-BAÑOS, F. J. 2007. Integrating XQuery and logic programming. In *Proceedings of the Workshop on Logic Programming*, University of Würzburg, Würzburg, Germany, 12 pages.

APT, K. R. 1990. Logic programming. In *Handbook of Theoretical Computer Science*, Vol. B: Formal Models and Semantics, J. van Leewen, Ed. MIT Press, Massachusetts Institute of Technology, USA, Chapter 10, 493–574.

ATANASSOW, F., CLARKE, D. AND JEURING, J. 2004. UUXML: A type-preserving XML schema haskell data binding. In *Proc. of Practical Aspects of Declarative Languages*. LNCS 3057. Heidelberg, Germany, 71–85.

BAILEY, J., BRY, F., FURCHE, T. AND SCHAFFERT, S. 2005. Web and semantic web query languages: A survey. In *Proc. of Reasoning Web, First International Summer School*, LNCS 3564. Heidelberg, Germany, 35–133.

BAUMGARTNER, R., FLESCA, S. AND GOTTLOB, G. 2001. The elog web extraction language. In *Proc. of International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LNCS 2250. Heidelberg, Germany, 548–560.

BENZAKEN, V., CASTAGNA, G. AND FRISH, A. 2005. CDuce: An XML-centric general-purpose language. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming*. ACM Press, New York, USA, 51–63.

BERNERS-LEE, T., HENDLER, J. AND LASSILA, O. 2001. 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.

BOLEY, H. 2000a. Relationships between logic programming and RDF. In *Proc. of Advances in Artificial Intelligence*, LNCS 2112. Heidelberg, Germany, 201–218.

BOLEY, H. 2000b. Relationships between logic programming and XML. In *Proc. of the Workshop on Logic Programming*, GMD Report 90. Würzburg, Germany, 19–34.

BOLEY, H. 2001. The rule markup language: RDF-XML data model, XML schema hierarchy, and XSL transformations. In *Proc. of International Conference on Applications of Prolog*. Prolog Association of Japan, Tokyo, Japan, 124–139.

BONCZ, P. A., GRUST, T., VAN KEULEN, M., MANEGOLD, S., RITTINGER, J. AND TEUBNER, J. 2005. Pathfinder: XQuery - the relational way. In *Proc. of the International Conference on Very Large Databases*. ACM Press, New York, USA, 1322–1325.

BORGIDA, A. 1996. On the relative expressiveness of description logics and predicate logics. *Artificial Intelligence 82,* 1-2, 353–367.

BRY, F. AND SCHAFFERT, S. 2002a. The XML query language Xcerpt: Design principles, examples, and semantics. In *Proc. of Web, Web-Services, and Database Systems*, LNCS 2593. Heidelberg, Germany, 295–310.

BRY, F. AND SCHAFFERT, S. 2002b. Towards a declarative query and transformation language for XML and semistructured data: Simulation unification. In *Proc. of International Conference on Logic Programming*, LNCS 2401. Heidelberg, Germany, 255–270.

CABEZA, D. AND HERMENEGILDO, M. 2001. Distributed WWW Programming using (Ciao-) prolog and the PiLLoW library. *Theory and Practice of Logic Programming 1,* 3, 251–282.

CHAMBERLIN, D. 2002. XQuery: An XML query language. *IBM Systems Journal* 41, 4, 597–615.

CHAMBERLIN, D., DRAPER, D., FERNÁNDEZ, M., KAY, M., ROBIE, J., RYS, M., SIMEON, J., TIVY, J. AND WADLER, P. 2004. *XQuery from the experts*. Addison Wesley, Boston, USA.

COELHO, J. AND FLORIDO, M. 2003. Type-based XML processing in logic programming. In *Proc. of the International Symposium on Practical Aspects of Declarative Languages*, LNCS 2562. Heidelberg, Germany, 273–285.

COELHO, J. AND FLORIDO, M. 2004. CLP(Flex): Constraint logic programming applied to XML processing. In *Proceedings of the CoopIS/DOA/ODBASE*, LNCS 3291. Heidelberg, Germany, 1098–1112.

DECKER, S., MELNIK, S., HARMELEN, F. V., FENSEL, D., KLEIN, M. C. A., BROEKSTRA, J., ERDMANN, M. AND HORROCKS, I. 2000. The Semantic Web: The Roles of XML and RDF. *IEEE Internet Computing* 4, 5, 63–74.

FERNÁNDEZ, M. AND SIMEON, J. 2003. Growing XQuery. In *Proc. of the Object-Oriented Programming, European Conference*, LNCS 2743. Heidelberg, Germany, 405–430.

FERNÁNDEZ, M., SIMEON, J. AND WADLER, P. 2000. An algebra for XML query. In *Proc. of Foundation of Software Technology and Theoretical Computer Science*, LNCS 1974. Heidelberg, Germany, 11–45.

GROSOF, B. N., HORROCKS, I., VOLZ, R. AND DECKER, S. 2003. Description logic programs: Combining logic programs with description logic. In *Proc. of the International Conference on World Wide Web*. ACM Press, USA, 48–57.

HORROCKS, I. AND PATEL-SCHNEIDER, P. F. 2004. A Proposal for an OWL rules language. In *Proc. of International Conference on World Wide Web*. ACM Press, New York, USA, 723–731.

HOSOYA, H. AND PIERCE, B. C. 2003. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology 3,* 2, 117–148.

MARIAN, A. AND SIMEON, J. 2003. Projecting XML documents. In *Proc. of International Conference on Very Large Databases*. Morgan Kaufmann, Burlington, USA, 213–224.

MAY, W. 2004. XPath-Logic and XPathLog: A logic-programming style XML data manipulation language. *Theory and Practice of Logic Programming 4,* 3, 239–287.

O'NEIL, P., O'NEIL, E., PAL, S., CSERI, I., SCHALLER, G. AND WESTBURY, N. 2004. OrdPaths: Insert-friendly XML node labels. In *Proc. of the ACM SIGMOD Conference*. ACM Press, New York, USA, 903–908.

RÉMY, D. 2002. *Applied Semantics: Advanced Lectures*, LNCS 2395. Heidelberg, Germany. Chapter Using, Understanding, and Unraveling the OCaml Language. From Practice to Theory and Vice Versa, 115–137.

SCHAFFERT, S. AND BRY, F. 2002. 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*. CEUR Workshop Proceedings 60, Aachen, Germany, 22 pages.

SEIPEL, D. 2002. Processing XML-documents in Prolog. In *Procs. of the Workshop on Logic Programming 2002*. Technische Universität Dresden, Dresden, Germany, 15 pages.

SIMEON, J. AND WADLER, P. 2003. The essence of XML. In *Proc. of SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, USA, 1–13.

TATARINOV, I., VIGLAS, S. D., BEYER, K., SHANMUGASUNDARAM, J., SHEKITA, E. AND ZHANG, C. 2002. Storing and querying ordered XML using a relational database system. In *Proc. of the ACM SIGMOD Conference*. ACM Press, New York, USA, 204–215.

THIEMANN, P. 2002. A typed representation for HTML and XML documents in Haskell. *Journal of Functional Programming 12,* 4&5, 435–468.

W3C. 2001. XML Schema 1.0. Tech. rep., www.w3.org.

W3C. 2004a. OWL Ontology Web Language. Tech. rep., www.w3.org.

W3C. 2004b. Resource Description Framework (RDF). Tech. rep., www.w3.org.

W3C. 2007a. Extensible Markup Language (XML). Tech. rep., www.w3c.org.

W3C. 2007b. XML Path Language (XPath) 2.0. Tech. rep., www.w3.org.

W3C. 2007c. XML Query Working Group and XSL Working Group, XQuery 1.0: An XML Query Language. Tech. rep., www.w3.org.

WADLER, P. 2002. XQuery: A typed functional language for querying XML. In *Advanced Functional Programming, International School*, LNCS 2638. Heidelberg, Germany, 188–212.

WALLACE, M. AND RUNCIMAN, C. 1999. Haskell and XML: Generic combinators or type-based translation? In *Proceedings of the International Conference on Functional Programming*. ACM Press, New York, USA, 148–159.

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

WOLZ, R. 2004. Web Ontology Reasoning with Logic Databases. Ph.D. thesis, Universität Fridericiana zu Karlsruhe.