Magic Sets for the XPath Language

Jesús M. Almendros-Jiménez
Dept. de Lenguajes y Computación
Universidad de Almería
jalmen@ual.es

Antonio Becerra-Terón
Dept. de Lenguajes y Computación
Universidad de Almería
abecerra@ual.es

Francisco J. Enciso-Baños
Dept. de Lenguajes y Computación
Universidad de Almería
fjenciso@ual.es

Abstract

The eXtensible Markup Language (XML) is considered as the format of choice for the exchange of information among various applications on the Internet. Since XML is emerging as a standard for data exchange, it is natural that queries among applications should be expressed as queries against data in XML format. This use gives rise to a requirement for a query language expressly designed for XML resources. World Wide Web Consortium (W3C) convened to create the XQuery language, concretely, a typed functional language for querying XML documents. One key aspect of the XQuery language is the use of the XPath language as basis for handling the structure of a XML document. In this paper, we present a proposal for 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 addition, we study how to query by means of the XPath language against a logic program representing a XML document. It evolves the specialization of the logic program with regard to the XPath expression. This specialization technique is based on the well-known transformation technique called Magic Sets and studied for deductive databases. The bottom-up evaluation of the specialized program is used for answering the query in the XPath language.

1 Introduction

The eXtensible Markup Language (XML) is considered as the format of choice for the exchange of information among various applications on the Internet. The use of tags makes XML data self-describing, and the extensible nature of XML makes it possible to define new kinds of documents for specialized purposes. As the importance of XML has increased, a number of standards has grown up around it, many of which were defined by the *World Wide Web Consortium* (W3C). For example, *XML Schema* [W3C01] provides a notation for defining new types of elements and documents; *XML Path Language* (XPath) [W3C04c] provides a notation for selecting elements within a XML document; and finally, *eXtensible StyleSheet Language Transformations* (XSLT) [W3C04e] provides a notation for transforming XML documents from one representation to another. Since XML is emerging as a standard for data exchange, it is natural that queries among applications should be expressed as queries against data in XML format. This use gives rise to a requirement for a query language expressly designed for XML resources. W3C convened to create the *XQuery language*, concretely, a typed functional language for querying XML documents [W3C04d, Wad02, Cha02]. One key aspect of the XQuery language is the use of the language XPath as basis for handling the structure of a XML document.

Logic-based languages have been proved useful in many areas since they allow to build small, declarative and extensible programs. For the database area, for instance *Datalog* has been investigated for querying and rule-based data manipulation. One of the key aspects of database languages based on logic pro-

gramming, Datalog among others, is the use of a *fixed-point operator* [Apt90] as evaluation mechanism, following a *bottom-up evaluation* of the program for query solving. For efficiency reasons, a *program specialization* technique called *Magic Sets* [BR91] is achieved with regard to a goal.

A *XML document* basically is a *labelled tree* with nodes representing *composed or non-terminal items* and leaves representing *values or terminal items*. The *XML schema*, which is also a XML document, defines the structure of well-formed documents and thus it can be seen as a type definition. Therefore, well-formedness analysis can be seen as type checking [SW03, HP03]. XPath considered as query language expresses a query against a XML document. Essential to semi-structured data [ABS00] is the selection of data from incompletely specified data items as in a 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. For instance, let us consider the following XML document:

Example of XML Document

```
<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>
```

representing a set of books, where each record stores the authors, titles and reviews for each book; and each record has an attribute representing the publishing year. 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

(1) `/books/book[author=Suciu]/title`

----

(2) `/books//title`

Expected XML Answer

(1) `<title> Data on the Web </title>`

----

(2) `<title> Data on the Web </title>`
`<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.

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:

- A 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 a 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.
- Our second contribution is that once XML documents can be described by means of a logic program, a XPath expression against the document requires to obtain a subset of the *Herbrand model* [Apt90] represented by the logic program. In deductive databases, the *bottom-up-based computation model* [BR91, ABS01] specializes a logic program with regard to a given query in order to compute the *subset of the Herbrand model* needed for answering the query. Our idea is to provide a *specialization program method*, but in this case, for handling of XPath expressions. Therefore, we will specialize the logic program representing a XML document with regard to a XPath expression in order to get the answer; that is, the XML data relevant to the query. This specialization technique is based on *Magic Sets* technique allowing a *bottom-up evaluation* of the specialized program in order to obtain the answer of the query.
- 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*). 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 bottom-up evaluation of such specialized program computes the answer.

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 type inference. However, we think that we might adapt our technique to directly translate XML schemas (or DTD's) into logic rules.

1.1 Related Work

In order to handle XML documents, some logic languages and formalisms have been proposed. For instance, XCERPT [SB02] 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 language, a specialized unification for query terms has been studied in [BS02]. The same can be said for XPathLog (LOPIX system) [May04], which is a Datalog-style extension of XPath with variable bindings. The Rule Markup Language (RULEML) [Bol01] translates Prolog facts and rules into XML documents allowing the combination of XML and RDF (Resource Description Framework) documents. Finally, some Prolog implementations include libraries for XML document loading and querying such as SWI-Prolog [Wie05] and CIAO [CH01].

In some of cited approaches, [SB02, May04] 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 traditional Prolog terms. It involves to define a translation of the structure of XML documents to Prolog terms. It is the same case as Prolog implementations for loading XML documents using a unique Prolog term for the entire XML document. However, we think that our translation is more refined than Prolog implementations, and suitable for a more efficient evaluation method.

With respect to the XPath queries, in the approaches [SB02, May04], the XPath expressions are translated into Prolog goals using the new query terms, and in the Prolog implementations, XPath is handled by using Prolog predicates. In our case, the XPath queries involve a Magic sets-based program transformation and bottom-up evaluation. One of the advantages of the bottom-up evaluation is to obtain sets of facts in each step of evaluation, which is called *"set-at-time evaluation"* in contraposition to the *"tuple-at-time evaluation"* of the traditional top-down evaluation method. It is suitable for databases once facts can be stored in secondary memory and therefore *minimizing disk accesses*. The bottom-up evaluation of the transformed program will obtain the answer by means of the reconstruction of the XML document representing the result from the set of obtained facts. The reconstruction assumes the same criteria as the translation of XML document-logic program.

With regard to [Bol01], 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 (for instance, ontologies) is handled by means of logic programming. In fact, our idea is to consider these aspects as future work. Our approach opens two promising research lines.

- The first one, the extending of XPath to a more powerful query language such as XQuery; 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 (XDUCE and GALAX) [HP03, MS03].
- The second one, the use of logic programming as inference engine for the so-

called "Semantic Web", by introducing RDF documents or OWL (Ontology Web Language) [W3C04b, W3C04a].

The structure of the paper is as follows. Section 2 will study the translation of XML documents into Prolog; Section 3 will discuss the magic-sets based technique applied to XPath queries; and finally, Section 4 will show the conclusions and future work.

2 Translating XML Documents into Prolog

In this section, we will show how to translate a XML document into a logic program. As commented in the introduction we will use a set of rules for describing the XML schema and a set of facts for storing the XML document. With this aim, we will show a set of general criteria for the building of the Prolog program from a XML document.

As running example, we will consider the following XML document representing a book database. In the XML document *tags* are used for specifying the structure of each XML element representing, in this case, a set of `books` described by means of the names of the authors, the title and a review. Each `book` is qualified by means of an *attribute* called `year`. For each *element* `book`, we have three grouped *sub-elements* `author`, `title` and `review`. In addition, the element `review` contains sub-elements used for formatting the text described by the review.

An Example of XML document:

```
<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>
```

Here, the XML database includes two books. The first one, edited in 2003, with `authors` `Abiteboul`, `Buneman` and `Suciu`, and `title` `'Data on the Web'`. Finally, the opinion of the reviewers for this book is: `A` *fine* `book`. The second one, edited in 2002, was written by `Buneman` with title `XML in Scotland`, and the opinion of the reviewers is *The best ever!*. In this XML document, we can see typical features of a semi-structured data model [ABS00]: heterogeneous records, in particular, non-first normal relations, missing values, among others. Once shown the example, now, we focus on the general criteria for translating XML documents into a logic program. These criteria can be summarized as follows:

1. Each tag (element) is translated into a predicate name. Each predicate has *three arguments*.
   - The first one is used for building a *prolog term* containing the XML document.
   - The second argument of the predicate is used for *numbering each node* (called *node number*) of the XML document tree (see Figure 1).
   - The third one is use for *numbering each type* (called *type number*) (see Figure 1).

   Next, we will explain the use of node and type number in the logic program.
2. For un-tagged elements grouped together tagged elements the predicate name `unlabeled` is used.
3. Each non-terminal tag is translated into a function symbol named as "name-element"+type with an argument for each (sub)element and an additional argument for storing the list of attributes.
4. Each terminal element is translated into a fact, numbered as in the figure 1. For instance, the above XML document can be represented by means of a logic program as follows:

Prolog program of a XML document

```
Rules (Schema):
--------------------------------
books(bookstype(A, []), L,1) :-
        book(A, [B|L],2).
book(booktype(A, B, C, [D]), L ,2) :-
        author(A, [E|L],3),
        title(B, [F|L],3),
        review(C, [G|L],3),
        year(D, L,3).
review(reviewtype(A,B,[]),L,3):-
        unlabeled(A,[J|L],4),
        em(B,[K|L],4).
review(reviewtype(A,[]),[I|L],3):-
        em(A,[J|L],5).
```

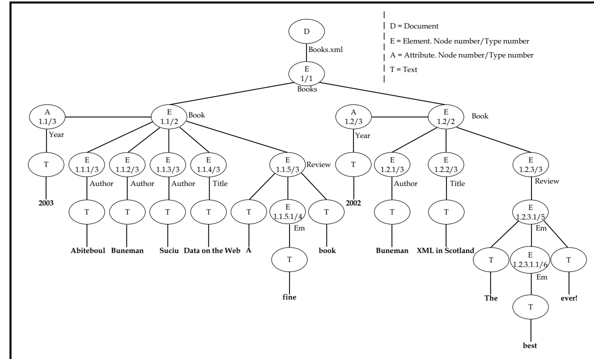

Figure 1: Node and type numbering in the tree structure of the above XML document

```
em(emtype(A,B,[]),L,5) :-
        unlabeled(A,[G|L],6),
        em(B, [H|L],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).
```

In this example, we can see the translation of each tag into a predicate name: `books`, `book`, etc. Each predicate has three arguments. The first one consists on a function symbol with the same name as the tag, but adding the suffix `type`, which encapsulates the XML document. Therefore, we have `bookstype`, `booktype`, etc. Each function symbol has arity $n+1$ where n is the number of subelements of the given element, and it has an extra-argument which is devoted to store the attribute list. The second argument is used for numbering each node. We have numbered the nodes of the XML tree by levels and from *left to right* starting from 1. 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 XML tree due to the use of lists for representing them. The last argument of the predicates is a number used for numbering each type. It will explained in the next point. Finally, each element, which is not labelled but grouped, is translated into a label called `unlabeled`. This is the case of "`A`" and "`book.`" in the first review.

5. In the case of a non-terminal does not always have the same structure, we introduce more than one rule with different type number for each kind of subelements. For instance, in the running example we have in each schema rule sequences of type numbers: 1-2, 2-3-3-3-3, 3-4-4, 3-5, 5-6-6. From these ones, the most significative ones are 3-4-4 and 3-5 which distinguish two cases for the type `review`. The first type is used in the first book where `fine` is emphasized but "`A`" and "`book.`" not. The second type is used in the second book where "`The best ever!`" is emphasized but "`best`" is doubly emphasized. The so-called *type number* is vital to distinguish each kind of type in the same element. Let us remark that the use of different type numbers for the same "type" as `review` can be justified due to the occurrence of different instances of the same type in different positions in the document and possibly with some missing val-

ues. However, when the instances occur in the same position (i.e. level of the XML tree) and with the same kind of values, they are numbered with the same type number.

6. Whenever there is more than one value for the same sub-tag in the same element, we introduce one fact for each value, numbered with the same type number, but distinct node number. For instance w.r.t. the running example:

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

   Let us remark that we could use a unique fact and use a Prolog list for storing the three elements. We believe that both choices can be sound, and our transformation technique could be adapted.

7. In the case of an element has several attributes, a Prolog list is used for storing them. For instance, with respect to the following document:

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

   we consider the following rule:

```
book(booktype(A, B, C, [D,J]), L, 2) :-
        author(A, [G|L],3),
        title(B, [H|L],3),
        review(C, [I|L],3),
        year(D,L,3),
        keyword(J,L,3).
```

8. In the case of recursively defined XML documents, they are handled by means of a recursive rule. For instance:

```
em(emtype(A,B,[]),L,5) :-
        unlabeled(A,[G|L],6),
        em(B, [H|L],6).
```

   This rule expresses that an emphasized text can include other(s) emphasized text(s).

9. In the case of using the XML schema, XML types can be identified with Prolog-style types whenever it is possible. In the case of handling of *lists of integers* and *strings*, we introduce *lists*. *Integers*, *Reals*, etc can be handled by means of the type *Integer*, *Real* of the host language distribution. For instance, consider the following XML document and its corresponding rule:

```
<book year="1999 2003">
<author>Abiteboul</author>
<title>Data on the Web</title>
<review>A <em>fine</em> book.</review>
</book>
----------------------
book(booktype(A, B, C, [Y]), L,2) :-
        author(A, [G|L],3),
        title(B, [H|L],3),
        review(C, [I|L],3),
        year(Y,L,3).
year(['1999','2003'],[1,1],1).
```

   The use of XML types enables to use more sophisticated queries by using binary operators as $\geq, \leq, mod, ...$ and other string and list operations. For simplicity, in this paper we have only considered values in the type string, but other kinds of types can be considered as well as operations on them. Finally, let us remark that by using the XML schema we can adopt the name used by a XML "`complextype`" element for naming function symbols; and also the "`mixed`" attribute can be translated into "unlabeled" predicate name.

3 Magic Set Transformation for XPath Expressions

In this section, we will present the magic set transformation technique for querying XPath expressions against a XML document represented by means of a logic program. With this aim, we will present the general criteria for the transformation technique.

3.1 Filtered rules

Since we use magic set transformations, firstly we need to add *magic filters* to each rule. For instance, with respect to our running example, we will consider the following set of rules:

```
Filtered Rules
---------------------------------
books(bookstype(A, []),L,1) :-
        mg_books(bookstype(A, []),L,1),
        book(A, [C|L],2).
book(booktype(A, B, C, [D]),L,2) :-
        mg_book(booktype(A, B, C, [D]),L,2),
        author(A, [G|L],3),
        title(B, [H|L],3),
        review(C, [I|L],3),
        year(D,L,3).
review(reviewtype(A,B,[]),L,3):-
        mg_review(reviewtype(A,B,[]),L,3),
        unlabeled(A,[J|L],4),
        em(B,[K|L],4).
```

```
review(reviewtype(A,[]),L,3):-
        mg_review(reviewtype(A,[]),L,3),
        em(A,[K|L],5).
em(emtype(A,B,[]),L,5) :-
        mg_em(emtype(A,B,[]),L,5),
        unlabeled(A,[G|L],6),
        em(B, [H|L],6).
Filtered Facts
---------------------------------
year('2003', [1, 1],3):- mg_year('2003', [1, 1],3).
author('Abiteboul', [1, 1, 1],3):-
        mg_author('Abiteboul', [1, 1, 1],3).
...
```

They are the so-called *filtered rules and facts* like typical magic-set based transformations.

3.2 Magic Transformation

Now, we present the general criteria for the specialization technique:

1. The handling of a XPath query involves the introduction of *one or more passing rules* and *one or more seeds*.
2. The seeds are generated from either the last element or the element situated before the first boolean condition on the XPath expression.
3. The passing rules are generated from the seed to terminal tags.

- For instance, we can assume a XPath query such as /books/book/author, requiring the authors in the book database. In this case, we have to generate the seed mg_author(X, Y, 3) where X, Y are logic variables, and the number 3 indicates the type number. The bottom-up evaluation of the filtered program from this seed will result on the following set of facts which represents the following XML document:

```
Computed Set of Facts
author('Abiteboul', [1, 1, 1],3).
author('Buneman', [2, 1, 1],3).
author('Suciu', [3, 1, 1],3).
author('Buneman', [1, 2, 1],3).
Represented XML Document
<result>
    <author>Abiteboul</author>
    <author>Buneman</author>
    <author>Suciu</author>
    <author>Buneman</author>
</result>
```

  In order to build the XML document, the idea is to consider the schema rules and the computed set of facts. In this case, author has no schema rules, and therefore the result can be built directly from the facts, considering the predicate name as tag and following the node numbering for the ordering of the XML elements.

- Now, we can assume the XPath expression /books/book. Now, the seed is mg_book(X, Y, 2), and since book has attributes and subelements, we need to generate the following *passing rules* which enable, from bottom-up evaluation, to generate, for each book, facts for author, title, review and year.

```
mg_author(A, [D|L] ,3) :-
        mg_book(booktype(A, E, F, [G]),L,2).
mg_review(A, [D|L],3) :-
        mg_book(booktype(E, F, A, [G]),L,2).
mg_title(A, [D|L],3) :-
        mg_book(booktype(E, A, F, [G]),L,2).
mg_year(A,L,3) :-
        mg_book(booktype(D, E, F, [A]),L,2).
mg_unlabeled(A,[J|L],4):-
        mg_review(reviewtype(A,B,[]),L,3).
mg_em(B,[K|L],4):-
        mg_review(reviewtype(A,B,[]),L,3).
mg_em(A,[K|L],5):-
        mg_review(reviewtype(A,[]),L,3).
mg_unlabeled(A,[G|L],6):-
        mg_em(emtype(A,B,[]),L,5).
mg_em(B, [H|L],6):-
        mg_em(emtype(A,B,[]),L,5).
```

  They are built as usual in magic-set based transformations but in this case *without considering a left-to-right information passing strategy.* The bottom-up evaluation of the above filtered rules together these passing rules from the seed mg_book(X, Y, Z, 2) is able to compute the following set of facts:

```
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('2003', [1, 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).
year('2002', [2, 1], 3).
review(reviewtype('A','fine',[]),[5, 1, 1],3).
review(reviewtype('book','fine',[]),[5, 1, 1],3).
review(reviewtype(emtype('The','best',[]),[]),
[3, 2, 1],3).
review(reviewtype(emtype('ever!','best',[]),[]),
[3, 2, 1],3).
em(emtype('The','best',[]),[1, 3, 2, 1],5).
em(emtype('ever!','best',[]),[1, 3, 2, 1],5).
```

  From this set of facts we can build the following answer in XML format:

```
<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>
```

In order to build this XML document, we need to consider the following subset of schema rules, including the schema from book (i.e. the last element in the XPath expression) up to terminal tags:

```
book(booktype(A, B, C, [D]),L,2) :-
          author(A, [G|L],3),
          title(B, [H|L],3),
          review(C, [I|L],3),
          year(D, L,3).
review(reviewtype(A,B,[]),L,3):-
          unlabeled(A,[J|L],4),
          em(B,[K|L],4).
review(reviewtype(A,[]),L,3):-
          em(A,[K|L],5).
em(emtype(A,B,[]), L,5) :-
          unlabeled(A,[G|L],6),
          em(B, [H|L],6).
```

These schema rules together with the generated facts allow the reconstruction of the XML document representing the result. It is easy to build a program for writing the XML document representing the result into a file. The ideas is to follow the path from the last element to the XPath query up to terminal elements rebuilding the original structure, following the schema rules. Let us remark that the last group of facts, in bold style, are only computed as auxiliary results and they are not needed for the reconstruction of the XML document.

- With respect to the generated seeds, in general a set of seeds is generated, concretely one for each element to be retrieved. Next, we show different examples of XPath expressions with their corresponding generated seeds.

XPath Expression	Generated Seed
(1) /books/book/**author**	**(1)** mg_author(X, Y, 3)
(2) /books/**book**	**(2)** mg_book(X, Y, 2)
(3) /books/book/*	**(3)** mg_author(X, Y, 3) **(3)** mg_title(X, Y, 3) **(3)** review(X, Y, 3) **(3)** mg_year(X, Y, 3)

In case (3), the XPath expression requests the subelements of book, and thus a seed for each one of them is generated.

- However whenever *a condition* is introduced, the seed is *forwarded* to the first occurrence of a condition. For instance, let us consider the XPath expression /books/book[author = Suciu]/title. In this case, we have a condition in the form of author = Suciu. Now, we have to generate the seed $\mathtt{mg_book}(\mathtt{booktype}('\mathtt{Suciu}', \mathtt{A}, \mathtt{B}), \mathtt{C}, 2)$. That is, the seed is forwarded to the element situated before of the first boolean condition, in this case book. In addition, mg_book is instantiated with Suciu in the function symbol booktype, and, of course, in the position representing the authors. In addition, we have to consider the following passing rules:

```
mg_author(A,[D|L],3) :-
          mg_book(booktype(A, E, F, [G]),L,2).
mg_title(A,[D|L],3) :-
          mg_book(booktype(E, A, F, [G]),L,2),
          author(E,[H|L],3).
```

In the bottom-up evaluation the seed will firstly trigger the retrieval of the author 'Suciu'. In particular, it retrieves the node numbers of Suciu's books. It is achieved due to the instantiation of the corresponding argument in the seed and the use of the first passing rule. Afterwards, it allows the retrieval of Suciu's book titles. With this aim, note that the predicate $\mathtt{author}(\mathtt{E}, [\mathtt{H}|\mathtt{L}], 3)$ has been included *as condition for the passing rules* corresponding to the element title. It ensures that *Suciu's book titles are the only computed.* The use of $\mathtt{author}(\mathtt{E}, [\mathtt{H}|\mathtt{L}], 3)$ is vital for efficient retrieval of such titles, given that the node number has been instantiated in this predicate in

the first step and the information is passed to the predicate of title in the second step. It corresponds, in some sense, with a *left-to-right information passing strategy*. In this case, the generated fact is author('Suciu', [3, 1, 1], 3) in which the node number [3, 1, 1] is used for retrieving the fact title('Data on the Web', [4, 1, 1], 3). Next, we show the computed facts by means of the bottom-up evaluation as well as the XML document represented by these facts:

```
Computed Set of Facts
author('Suciu', [3,1,1],3).
title('Data on the Web', [4,1,1],3).
Represented XML Document
<result>
        <title>Data on the Web</title>
</result>
```

Let us remark that there is an additional computed fact: author('Suciu', [3, 1, 1], 3), which is not used for the building of the resulting XML document, but that it has been used for computing the relevant fact. The XML document can be directly built from the fact, given that title has no schema rules.

- In the case of *two or more conditions*, such as /books/book[@year = 2002 and title = Data on the Web]/author, we have to consider the seed mg_book (booktype(A,' Data on the Web', C, ['2002']), L, 2), as well as the following passing rules:

```
mg_year(A, L,3) :-
        mg_book(booktype(E, F, G, [A]), L,2).
mg_title(A, [D|L],3) :-
        mg_book(booktype(E, A, F, [G]),L,2),
        year(G,L,3).
mg_author(A, [D|L],3) :-
        mg_book(booktype(A, E, F, [G]),L,2),
        year(G,L,3),
        title(G,[E|L],3).
```

In the passing rules, we follow a *left to right* strategy of the boolean condition. That is, starting from the seed mg_book(booktype(A,' Data on the Web', C, ['2002']), L, 2), firstly the first passing rule triggers the retrieval of books for this year 2002. Afterwards, the second passing rule triggers the retrieval of titles of this year (using the node number instantiated in the previous step); concretely book titled "Data on the Web". Finally, the last passing rule retrieves the author of such books, using node numbers instantiated in the previous steps.

- Whenever the occurrences of conditions are *at different level of the XPath query*, such as the XPath expression /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>
```

then the seed is mg_book(booktype (authortype(A,' Serge', B), C, D, ['2002']), L, 2) and the passing rules are as follows:

```
mg_year(A,L,3) :-
        mg_book(booktype(E, F, G, [A]),L,2).
mg_author(A, [D|L],3) :-
        mg_book(booktype(A, E, F, [G]),L,2),
        year(G,L,3).
mg_name(A, [E|L],4) :-
        mg_author(authortype(A,G,[]),L,3).
```

Here, firstly, we filter the books by year, next we retrieve authors for these books, and finally author names are filtered and recovered.

XPath is a rich query language with many variants. The main cases have been shown. The handling of the rest of cases involves modifications on the number of seeds and the form of passing rules.

4 Conclusions and Future Work

In this paper, we have presented how to represent XML documents by means of logic programming. We have studied how to transform such program by means of magic set transformations in order to obtain the answers w.r.t. a XPath query. As future work, we will study how to extend our work in order to translate a XQuery query into logic programming.

References

[ABS00] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML.* M. Kaufmann, 2000.

[ABS01] J. M. Almendros-Jiménez, A. Becerra-Terón, and J. Sánchez-Hernández. A Computational Model for Funtional Logic Deductive Databases. In *Proc. of ICLP*, LNCS 2237, pp. 331–347. Springer, 2001.

[Apt90] K. R. Apt. Logic programming. In *Handbook of Theoretical Computer Science*, chapter 10, pp. 493–574. MIT Press, 1990.

[Bol01] H. Boley. The Rule Markup Language: RDF-XML Data Model, XML Schema Hierarchy, and XSL Transformations. In *Proc. of INAP*, pp. 124–139. Prolog Association of Japan, 2001.

[BR91] C. Beeri and R. Ramakrishnan. On the Power of Magic. *JLP*, 10(3,4):255–299, 1991.

[BS02] F. Bry and S. Schaffert. Towards a Declarative Query and Transformation Language for XML and Semistructured Data: Simulation Unification. In *Proc. of ICLP*, LNCS 2401, pp. 255–270. Springer, 2002.

[CH01] D. Cabeza and M. Hermenegildo. Distributed WWW Programming using (Ciao-)Prolog and the PiLLoW Library. *TPLP*, 1(3):251–282, 2001.

[Cha02] D. Chanberlin. XQuery: An XML Query Language. *IBM Systems Journal*, 41(4):597–615, 2002.

[HP03] H. Hosoya and B. C. Pierce. XDuce: A Statically Typed XML Processing Language. *TOIT*, 3(2):117–148, 2003.

[May04] W. May. XPath-Logic and XPathLog: A Logic-Programming Style XML Data Manipulation Language. *TPLP*, 4(3):239–287, 2004.

[MS03] A. Marian and J. Simeon. Projecting XML Documents. In *Proc. of VLDB*, pp. 213–224. Morgan Kaufmann, 2003.

[SB02] S. Schaffert and F. Bry. A Gentle Introduction to Xcerpt, a Rule-based Query and Transformation Language for XML. In *Proc. of RuleML*, 2002.

[SW03] J. Simeon and P. Wadler. The Essence of XML. In *Proc. of POPL*, volume 38, pp. 1–13. ACM, 2003.

[W3C01] W3C. XML Schema 1.0. Technical report, www.w3.org, 2001.

[W3C04a] W3C. OWL Ontology Web Language. Technical report, www.w3.org, 2004.

[W3C04b] W3C. Resource Description Framework (RDF). Technical report, www.w3.org, 2004.

[W3C04c] W3C. XML Path Language (XPath) 2.0, Draft. Technical report, www.w3.org, 2004.

[W3C04d] W3C. XQuery 1.0: An XML Query Language. Technical report, www.w3.org, 2004.

[W3C04e] W3C. XSL Transformations (XSLT) Version 2.0. Technical report, www.w3.org, 2004.

[Wad02] P. Wadler. XQuery: A Typed Functional Language for Querying XML. In *AFP*, LNCS 2638, pp. 188–212. Springer, 2002.

[Wie05] J. Wielemaker. SWI-Prolog SGML/XML Parser, Version 2.0.5. Technical report, Human Computer-Studies (HCS), University of Amsterdam, March 2005.