Annotating "Fuzzy Chance Degrees" when Debugging XPath Queries

Jesús M. Almendros-Jiménez[1]*, Alejandro Luna[2], and Ginés Moreno[2]

[1] Dep. of Informatics, University of Almería, Spain
Email: jalmen@ual.es
[2] Dep. of Computing Systems, University of Castilla-La Mancha, Spain
Emails: Alejandro.Luna@alu.uclm.es, Gines.Moreno@uclm.es

Abstract. In this paper we present a method for debugging XPath queries which has been implemented with the fuzzy logic language MALP by using the FLOPER tool developed in our group. We describe how XPath expressions can be manipulated for obtaining a set of alternative queries matching a given XML document. For each new proposed query, we give a "chance degree" that represents an estimation on its deviation w.r.t. the initial expression. Our work is focused on providing to the programmers a repertoire of paths which can be used to retrieve answers.

Keywords: XPath; Fuzzy (Multi-adjoint) Logic Programming; Debugging

1 Introduction

The eXtensible Markup Language (XML) is widely used in many areas of computer software to represent machine readable data. XML provides a very simple language to represent the structure of data, using tags to label pieces of textual content, and a tree structure to describe the hierarchical content. XML emerged as a solution to data exchange between applications where tags permit to locate the content. XML documents are mainly used in databases. The XPath language [1] was designed as a query language for XML in which the path of the tree is used to describe the query. XPath expressions can be adorned with boolean conditions on nodes and leaves to restrict the number of answers of the query. XPath is the basis of a more powerful query language (called XQuery) designed to join multiple XML documents and to give format to the answer.

In spite of the simplicity of the XPath language, the programmer usually makes mistakes describing the path in which the data are allocated. Tipically, the programmer omits some of the tags of the path, adds more than necessary, and also uses similar but wrong tag names. When the query does not match to the tree structure of the XML tree, the answer is empty. However, we can also

* The author's work has been supported by the Spanish Ministry MICINN and Ingenieros Alborada IDI under grant TRA2009-0309, and the Junta de Andalucía (proyecto de excelencia) under grant TIC-6114.

find the case in which the query matches to the XML tree but the answer does not satisfy the programmer. Due to the inherent flexibility of XML documents, the same tag can occurs at several positions, and the programmer could find answers that do not correspond to her (his) expectations. In other words, (s)he finds a correct path, but a wrong answer. We can also consider the case in which a boolean condition is wrong, expressing a wrong range, and several boolean conditions that do not hold at the same time. When the programmer does not find the answer is looking for, there is a mechanism that can try to debug the query. In XPath there exists an operator, denoted by '//', that permits to look for the tag from that position. However, it is useless when the tag is present at several positions, since even though the programmer finds answers, does not know whether they are close to her (his) expectations.

XPath debugging has to take into account the previous considerations. Particularly, there is an underlying notion of *chance degree*. When the programmer makes mistakes, the number of bugs can be higher or lower, and the chance degree is proportional to them. Moreover, there are several ways on which each bug can be solved, and therefore the chance degree is also dependent from the number of solutions for each bug, and the quality of each solution. The quality of a solution describes the number of changes to be made. Finally, there is a case in which we have also focused our work. The case in which the mistake comes from a similar but wrong used tag. Here, the chance degree comes from the semantic closeness of the used tag.

Our proposed XPath debugging technique is guided by the programmer that initially establishes a value (i.e., a real value between 0 and 1), used by the debugger to penalize bugs in a proportional way. Additionally, we assume that the debugger is equipped with a table of similarities, that is, pairs of similar words with an assigned value in the range [0..1]. It makes possible that chance degrees be computed from similarity degrees.

The debugger reports a set of annotated paths by using an extended XPath syntax incorporating three annotations: JUMP, SWAP and DELETE. JUMP is used to represent that some tags have been added to the original expression, SWAP is used to represent that a tag has been changed by another one and, DELETE is used to represent that a tag has been removed. Moreover, the reported XPath expressions updates the original XPath expression, that is: case JUMP incorporates '//' at the position in which the bug is found; case SWAP includes the new tag; and finally case DELETE removes the wrong tag.

Additionally, our proposal permits the programmer to test the reported XPath expressions. The annotated XPath expressions can be executed in our tool (`http://dectau.uclm.es/fuzzyXPath/`) in order to obtain a ranked set of answers w.r.t. the chance degree. It facilitates the process of debugging because programmers can visualize answers to each query in a very easy way. Our implementation has been developed on top of the recently proposed fuzzy XPath extension [2, 3], which uses *fuzzy logic programming* to provide a fuzzy taste to XPath expressions. The implementation has been coded with the fuzzy logic programming

language MALP and developed with the FLOPER tool designed in our research group and freely accessible from `http://dectau.uclm.es/floper/`.

Although our approach can be applied to standard (crisp) XPath expressions, chance degrees in XPath debugging fits well with our proposal. Particularly, XPath debugging annotations can be seen as annotations of XPath expressions similary to the proposed DEEP and DOWN of fuzzy XPath [2, 3]. DEEP and DOWN serve to annotate XPath expressions and to obtain a ranked set of answers depending on they occur, more deeply and from top to down. Each answer is annotated with a *RSV (Retrieval Status Value)* which describes the degree of satisfaction of the answer. Here JUMP, SWAP and DELETE penalize the answers of annotated XPath expressions. DEEP and JUMP have, in fact, the same behavior: JUMP proportionally penalizes answers as deep as they occur. Moreover, in order to cover with SWAP, we have incorporated to our framework similarity degrees. Finally, let us remark that the current work is an extension of our previous published work [4].

The structure of the paper is as follows. After summarizing in Section 2 our fuzzy extension of XPath [2, 3], in Section 3 we describe our debugging technique. Implementation details are drawn in Section 4.

2 Fuzzy XPath

In this section we summarize the main elements of our proposed fuzzy XPath language described in [2, 3]. We firstly incorporate two structural constraints called DOWN and DEEP to which a certain degree of relevance is associated. So, whereas DOWN provides a ranked set of answers depending on the path they are found from "top to down" in the XML document, DEEP provides a ranked set of answers depending on the path they are found from "left to right" in the XML text. Both structural constraints can be used together, assigning importance's degrees with respect to the distance to the root XML element.

Secondly, our fuzzy XPath incorporates fuzzy variants of *and* and *or* for XPath conditions. Crisp *and* and *or* operators are used in standard XPath over boolean conditions, and enable to impose boolean requirements on the answers. XPath boolean conditions can be referred to attribute values and node content, in the form of equality and range of literal values, among others. However, the *and* and *or* operators applied to two boolean conditions are not precise enough when the programmer does not give the same value to both conditions. For instance, some answers can be discarded when they could be of interest by the programmer, and accepted when they are not of interest. Besides this, programmers would need to know in which sense a solution is better than another. When several boolean conditions are imposed on a query, each one contributes to satisfy the programmer's preferences in a different way and perhaps, the programmer's satisfaction is distinct for each solution.

We have enriched the arsenal of operators of XPath with fuzzy variants of *and* and *or*. Particularly, we have considered three versions of *and*: *and+*, *and*, *and-* (and the same for *or* : *or+*, *or*, *or-*) which make more flexible the composition of

fuzzy conditions. Three versions for each operator that come for free from our adaptation of fuzzy logic to the XPath paradigm.

One of the most known elements of fuzzy logic is the introduction of fuzzy versions of classical boolean operators. *Product*, *Łukasiewicz* and *Gödel* fuzzy logics are considered as the most prominent logics and give a suitable semantics to fuzzy operators. Our contribution is now to give sense to fuzzy operators into the XPath paradigm, and particularly in programmer's preferences. We claim that in our work the fuzzy versions provide a mechanism to force (and debilitate) conditions in the sense that stronger (and weaker) programmer preferences can be modeled with the use of stronger (and weaker) fuzzy conditions. The combination of fuzzy operators in queries permits to specify a ranked set of fuzzy conditions according to programmer's requirements.

Furthermore, we have equipped XPath with an additional operator that is also traditional in fuzzy logic: the average operator *avg*. This operator offers the possibility to explicitly give weight to fuzzy conditions. Rating such conditions by *avg*, solutions increase its weight in a proportional way. However, from the point view of the programmer's preferences, it forces the programmer to quantify his(er) wishes which, in some occasions, can be difficult to measure. For this reason, fuzzy versions of *and* and *or* are better choices in some circumstances.

Finally, we have equipped our XPath based query language with a mechanism for thresholding programmer's preferences, in such a way that programmer can request that requirements are satisfied over a certain percentage.
The proposed fuzzy XPath is described by the following syntax:

xpath	:=	['**[**'deep-down'**]**']path
path	:=	literal | text() | node | **@**att | node/path | node//path
node	:=	QName | QName[cond]
cond	:=	xpath op xpath | xpath num-op num
deep	:=	DEEP=num
down	:=	DOWN=num
deep-down	:=	deep | down | deep '**;**' down
num-op	:=	$> \mid = \mid < \mid <>$
fuzzy-op	:=	**and** | **and+** | **and-** | **or** | **or+** | **or-** | **avg** | **avg**{num,num}
op	:=	num-op | fuzzy-op

Basically, our proposal extends XPath as follows:

- **Structural constraints**. A given XPath expression can be adorned with «[DEEP $= r_1$; DOWN $= r_2$]» which means that the *deepness* of elements is penalized by r_1 and that the *order* of elements is penalized by r_2, and such penalization is proportional to the distance (i.e., the length of the branch and the weight of the tree, respectively). In particular, «[DEEP $= 1$; DOWN $= r_2$]» can be used for penalizing only w.r.t. document order. DEEP works for //, that is, the deepness in the XML tree is only computed when descendant nodes are explored, while DOWN works for both / and //. Let us remark that DEEP and DOWN can be used several times on the main *path* expression and/or any other *sub-path* included in conditions.

Fig. 1. Fuzzy Logical Operators

$\&_{\mathtt{P}}(x,y) = x * y$	$\|_{\mathtt{P}}(x,y) = x + y - x * y$	*Product: and/or*
$\&_{\mathtt{G}}(x,y) = \min(x,y)$	$\|_{\mathtt{G}}(x,y) = \max(x,y)$	*Gödel: and+/or-*
$\&_{\mathtt{L}}(x,y) = \max(x+y-1,0)$	$\|_{\mathtt{L}}(x,y) = \min(x+y,1)$	*Łuka.: and-/or+*

Fig. 2. Input XML document in our examples

```
<bib>
    <name>Classic Literature</name>
    <book year="2001" price="45.95">
        <title>Don Quijote de la Mancha</title>
        <author>Miguel de Cervantes Saavedra</author>
        <references>
            <novel year="1997" price="35.99">
                <name>La Galatea</name>
                <author>Miguel de Cervantes Saavedra</author>
                <references>
                    <book year="1994" price="25.99">
                        <title>Los trabajos de Persiles y Sigismunda</title>
                        <author>Miguel de Cervantes Saavedra</author>
                    </book>
                </references>
            </novel>
        </references>
    </book>
    <novel year="1999" price="25.65">
        <title>La Celestina</title>
        <author>Fernando de Rojas</author>
    </novel>
</bib>
```

– **Flexible operators in conditions**. We consider three fuzzy versions for each one of the classical conjunction and disjunction operators (also called connectives or aggregators) describing *pessimistic*, *realistic* and *optimistic* scenarios, see Figure 1. In XPath expressions the fuzzy versions of the connectives make harder to hold boolean conditions, and therefore can be used to debilitate/force boolean conditions. Furthermore, assuming two given *RSV*'s r_1 and r_2, the *avg* operator is obviously defined with a fuzzy taste as $(r_1 + r_2)/2$, whereas its *priority-based* variant, i.e. $avg\{p_1, p_2\}$, is defined as $(p_1 * r_1 + p_2 * r_2)/p_1 + p_2$.

In general, a fuzzy XPath expression defines, w.r.t. an XML document, a sequence of subtrees of the XML document where each subtree has an associated RSV. XPath conditions, which are defined as fuzzy operators applied to XPath expressions, compute a new RSV from the RSVs of the involved XPath expressions, which at the same time, provides a RSV to the node. In order to illustrate these explanations, let us see some examples of our proposed fuzzy version of XPath according to the XML document shown of Figure 2.

Example 1. Let us consider the fuzzy XPath query of Figure 3 requesting *title*'s penalizing the occurrences from the document root by a proportion of 0.8 and 0.9 by nesting and ordering, respectively, and for which we obtain the file listed

Fig. 3. Execution of the query «/bib[DEEP=0.8;DOWN=0.9]//title»

Document	**RSV computation**
<result>	
<title **rsv**="0.8000">Don Quijote de la Mancha</title>	$0.8000 = 0.8$
<title **rsv**="0.7200">La Celestina</title>	$0.7200 = 0.8 * 0.9$
<title **rsv**="0.2949">Los trabajos de Persiles y ...</title>	$0.2949 = 0.8^5 * 0.9$
</result>	

Fig. 4. Execution of the query «//book[@year<2000 avg{3,1} @price<50]/title»

Document	**RSV computation**
<result>	
<title **rsv**="1.00">Los trabajos de Persiles y ....</title>	$1.00 = (3 * 1 + 1 * 1)/(3 + 1)$
<title **rsv**="0.25">Don Quijote de la Mancha</title>	$0.25 = (3 * 0 + 1 * 1)/(3 + 1)$
</result>	

Fig. 5. Execution of the query «/bib[DEEP=0.5]//book[@year<2000 avg{3,1} @price<50]/title»

Document	**RSV computation**
<result>	
<title **rsv**="0.25">Don Quijote de la Mancha</title>	$0.25 = (3 * 0 + 1 * 1)/(3 + 1)$
<title **rsv**="0.0625">Los trabajos de ...</title>	$0.0625 = 0.5^4 * (3 * 1 + 1 * 1)/(3 + 1)$
</result>	

in Figure 3. In such document we have included as attribute of each subtree, its corresponding RSV. The highest RSVs correspond to the main *books* of the document, and the lowest RSVs represent the *books* occurring in nested positions (those annotated as related *references*).

Example 2. Figure 4 shows the answer associated to a search of books, possibly referenced *directly or indirectly* from other books, whose publishing year and price are relevant but the year is three times more important than the price. Finally, in Figure 5 we combine both kinds of (structural/conditional) operators, and the ranked list of solutions is reversed, where "*Don Quijote*" is not penalized with DEEP.

3 Debugging XPath

In this section we propose a debugging technique for XPath expressions. Our debugging process accepts as inputs a query Q preceded by the [DEBUG=r] command, where r is a real number in the unit interval. For instance, «[DEBUG=0.5] /bib/book/title».

Assuming an input XML document like the one depicted in Figure 2, the debugging produces a set of alternative queries $Q_1, ..., Q_n$ packed into an output XML document with the following structure (see also Figure 6):

<result>
<query cd="r_1" $attributes_1$> Q_1 </query>
...
<query cd="r_n" $attributes_n$> Q_n </query>
</result>

Fig. 6. Debugging query «[DEBUG=0.5]/bib/book/title»

```
<result>
  <query cd="1.0">/bib/book/title</query>
  <query cd="0.8" book="novel">/bib/[SWAP=0.8]novel/title</query>
  <query cd="0.5" book="//">/bib/[JUMP=0.5]//title</query>
  <query cd="0.5" bib="//">/[JUMP=0.5]//book/title</query>
  <query cd="0.45" book="" title="name">/bib/[DELETE=0.5][SWAP=0.9]name</query>
  <query cd="0.4" bib="//" book="novel">/[JUMP=0.5]//[SWAP=0.8]novel/title</query>
  <query cd="0.25" book="" title="//">/bib/[DELETE=0.5][JUMP=0.5]//title</query>
  <query cd="0.25" book="//" book="">/bib/[JUMP=0.5]//[DELETE=0.5]title</query>
  <query cd="0.25" bib="" book="//">/[DELETE=0.5][JUMP=0.5]//book/title</query>
  <query cd="0.25" bib="//" book="//">/[JUMP=0.5]//[JUMP=0.5]//title</query>
  <query cd="0.25" bib="//" bib="">/[JUMP=0.5]//[DELETE=0.5]book/title</query>
  <query cd="0.225" title="//" title="//" title="name">
          /bib/book/[JUMP=0.5]//[JUMP=0.5]//[SWAP=0.9]name</query>
  <query cd="0.225" bib="" book="//" title="name">
         /[DELETE=0.5][JUMP=0.5]//[SWAP=0.9]name</query>
  <query cd="0.225" bib="//" book="" title="name">
         /[JUMP=0.5]//[DELETE=0.5][SWAP=0.9]name</query>
  <query cd="0.2" bib="" book="//" book="novel">
          /[DELETE=0.5][JUMP=0.5]//[SWAP=0.8]novel/title</query>
   .........
</result>
```

where the set of alternative queries is ordered with respect to the CD key. This value measures the chance degree of the original query with respect to the new one, in the sense that as much changes are performed on Q_i and as more *traumatic* they are with respect to Q, then the CD value becomes lower.
In Figure 6, the first alternative, with the highest CD, is just the original query, thus, the CD is *1*, whose further execution should return «Don Quijote de La Mancha». Our debugger runs even when the set of answers is not empty, like in this case. The remaining options give different CD's depending on the chance degree, and provide XPath expressions annotated with JUMP, DELETE and SWAP commands.

In order to explain the way in which our technique generates the attributes and content of each *query* tag in the output XML debugging document, let us consider a generic path Q of the form: «[DEBUG=r]$/tag_1/.../tag_i/tag_{i+1}/...$», where we say that tag_i is at level i in the original query. So, assume that when exploring the input query Q and the input XML document D, we find that tag_i in Q does not occurs at level i in (a branch of) D. Then, we consider the following three situations.

3.1 Swapping Case

Instead of tag_i, we find tag'_i at level i in the input XML document D, where tag_i and tag'_i are two similar terms with similarity degree s. Then, we generate an alternative query by adding the attribute tag_i ="tag'_i" and replacing in the original path the occurrence "tag_i/" by "[SWAP= s]tag'_i/". The second query proposed in Figure 6 illustrates this case:

« <query cd="0.8" book="novel">/bib/[SWAP=0.8]novel/title</query> »

Let us observe that : 1) we have included the attribute «book="novel"» in order to suggest that instead of looking now for a *book*, finding a *novel* should be also a good alternative, 2) in the path we have replaced the tag *book* by *novel* and we have appropriately annotated the exact place where the change has been performed with the annotation [SWAP=0.8] and 3) the CD of the new query has been adjusted with the *similarity degree* 0.8 of the exchanged tags.

Now, it is possible to launch with our *FuzzyXPath* interpreter, the execution of the (fuzzy) XPath queries «/bib/novel/title» and «/bib/[SWAP=0.8]novel/title». In both cases we obtain the same result, i.e., «La Celestina» but with different RSV (or *Retrieval Status Value*): 1 and 0.8, respectively.

3.2 Jumping Case

Even when tag_i is not found at level i in the input XML document D, tag_{i+1} appears at a deeper level (i.e., greater than i) in a branch of D. Then, we generate an alternative query by adding the attribute tag_i=*"//"*, which means that tag_i has been jumped, and replacing in the path the occurrence *"tag_ i/"* by "[JUMP=r]//", where r is the value associated to DEBUG.

This situation is illustrated by the third and fourth queries in Figure 6, where we propose to jump tags *book* and *bib*. The execution of the queries returns different results, and as more tags are jumped, the resulting CD's become lower. Let us see the results of «/bib/[JUMP=0.5]//title» and «/[JUMP=0.5]//book/title», respectively:

```
<result>
  <title rsv="0.5">Don Quijote de la Mancha</title>
  <title rsv="0.5">La Celestina</title>
  <title rsv="0.03125">Los trabajos de Persiles y Sigismunda</title>
</result>

<result>
  <title rsv="0.5">Don Quijote de la Mancha</title>
  <title rsv="0.03125">Los trabajos de Persiles y Sigismunda</title>
</result>
```

3.3 Deletion Case

This scenario emerges when at level i in the input XML document D, we found tag_{i+1} instead of tag_i. So, the intuition tell us that tag_i should be removed from the original query Q and hence, we generate an alternative query by adding the attribute tag_i=*""* and replacing in the path the occurrence *"tag_ i/"* by "[DELETE=r]", being r the value associated to DEBUG.

This situation is illustrated by the fifth query in Figure 6, where the deletion of the tag *book* is followed by a swapping of similar tags *title* and *name*. The CD *0.45* associated to this query is defined as the *product* of the values

associated to both DELETE (*0.5*) and SWAP (*0.9*), and hence the chance degree of the original one is lower than the previous examples. So, the execution of query «/bib/[DELETE=0.5][SWAP=0.9]name», should produce the XML-based output:

```
<result>
  <name rsv="0.45">Classic Literature</name>
</result>
```

As we have seen in the previous example, the combined use of one or more debugging commands (SWAP, JUMP and DELETE) is not only allowed but also frequent. In other words, it is possible to find several debugging points.

When executing a query like «/[DELETE=0.5][JUMP=0.5]//[SWAP=0.9]name» , with several changes on its body (w.r.t. the original goal) and a CD *0.225* quite low, the RSV of the result is low too, since it has been obtained by multiplying the three values associated to the deletion of the tag *bib* (*0.5*), jumping the tag *book* (*0.5*) and the swapping of *title* by *name* (*0.9*):

```
<result>
  <name rsv="0.225">Classic Literature</name>
  <name rsv="0.028125">La Galatea</name>
</result>
```

It is important to note that the wide range of alternatives proposed by our technique (Figure 6 is still incomplete), reveals its high level of flexibility: programmers are free to use the alternative queries to execute them, and to inspect results up to their intended expectations.

Finally, we would like to remark that even when we have worked with a very simple query with three tags in our examples, our technique works with more complex queries with large paths and connectives in boolean conditions, as well as DEBUG used in several places on the query. For instance, in Figure 7, we show the result of debugging the following query: «[DEBUG=0.7]/bib/[DEBUG=0.6]book/[DEBUG=0.5]title».

4 Some Implementation Hints with MALP and FLOPER

In this section we assume familiarity with logic programming and its most popular language Prolog [5], for which MALP [6] (*Multi-Adjoint Logic Programming* [3]) allows a wide repertoire of *fuzzy connectives* connecting atoms in the bodies of clauses. Although the core of our application is written with (fuzzy) MALP rules, our implementation is based on the following items:

1. We have reused/adapted several modules of our previous Prolog-based implementation of (crisp) XPath described in [9, 10].
2. We have used the SWI-Prolog library for loading and writing XML files, in order to represent a XML document with Prolog term[4].

[3] See also [7, 8] and visit `http://dectau.uclm.es/floper` for downloading our FLOPER system.

[4] The notion of *term* (i.e., data structure) is just the same in MALP as in Prolog.

Fig. 7. Debugging of the query «[DEBUG=0.7]/bib/[DEBUG=0.6]book/[DEBUG=0.5]title»

```
<result>
  <query cd="1.0">/bib/book/title</query>
  <query cd="0.8" book="novel">/bib/[SWAP=0.8]novel/title</query>
  <query cd="0.7" bib="//">/[JUMP=0.7]//book/title</query>
  <query cd="0.6" book="//">/bib/[JUMP=0.6]//title</query>
  <query cd="0.56" bib="//" book="novel">/[JUMP=0.7]//[SWAP=0.8]novel/title</query>
  <query cd="0.54" book="" title="name">/bib/[DELETE=0.6][SWAP=0.9]name</query>
  <query cd="0.42" bib="" book="//">/[DELETE=0.7][JUMP=0.6]//book/title</query>
  <query cd="0.42" bib="//" book="//">/[JUMP=0.7]//[JUMP=0.6]//title</query>
  <query cd="0.378" bib="" book="//" title="name">
                    /[DELETE=0.7][JUMP=0.6]//[SWAP=0.9]name</query>
  <query cd="0.378" bib="//" book="" title="name">
                    /[JUMP=0.7]//[DELETE=0.6][SWAP=0.9]name</query>
  <query cd="0.336" bib="" book="//" book="novel">
                    /[DELETE=0.7][JUMP=0.6]//[SWAP=0.8]novel/title</query>
  <query cd="0.3" book="" title="//">/bib/[DELETE=0.6][JUMP=0.5]//title</query>
  <query cd="0.2646" bib="//" bib="" book="" title="name">
                    /[JUMP=0.7]//[DELETE=0.7][DELETE=0.6][SWAP=0.9]name</query>
  .........
</result>
```

Fig. 8. A Prolog term representing a XML document

```
[element(bib,[],
   [element(book,[year=2001,price=45.95],
            [element(title,[],[Don Quijote de la Mancha]),
            element(author,[],[Miguel de Cervantes Saavedra]),
            element(publications,[],
                        [element(book,[year=1997,price=35.99],
                                  [element(title,[],[La Galatea]),
                                  element(author,[],[Miguel de Cervantes Saavedra]),
                              element(publications,[],...])...]),])])
```

3. The parser of XPath has been extended to recognize new keywords such as DEBUG and others like DEEP, DOWN, *avg*, etc. with their proper arguments.
4. Each tag is represented as a data-term of the form: `element(Tag, Attributes, Subelements)`, where `Tag` is the name of the XML tag, `Attributes` is a Prolog list containing the attributes, and `Subelements` is a Prolog list containing the sub-elements (i.e. subtrees) of the tag. For instance, the document of Figure 2 is represented in SWI-Prolog like in Figure 8.
5. A predicate called `fuzzyXPath` where `fuzzyXPath(+ListXPath,+Tree,+Deep,+Down)` receives four arguments: (1) `ListXPath` is the Prolog representation of an XPath expression; (2) `Tree` is the term representing an input XML document and (3) DEEP/DOWN.
6. The evaluation of the query generates a *truth value* which has the form of a tree, called *tv tree*. Basically, the `fuzzyXPath` predicate traverses the Prolog tree representing a XML document annotating into the *tv tree* the corresponding DEEP/DOWN values. These actions directly revert on the new predicate `debugQuery` implementing the ideas described in this work.
7. Finally, the *tv tree* is used for computing the output of the query, by multiplying the recorded values. A predicate called `tv_to_elem` has been implemented to output the answer in a pretty way.

Fig. 9. Example of a XML output in *MALP*

```
tv(0.9,[[],
   tv(0.9,[element(title,[],[Don Quijote de la Mancha]),[],
   tv(1,[[],[],
   tv(1,[[],
      tv(0.9,[[],
         tv(0.9,[element(title,[],[La Galatea]),[],
         tv(1,[[],[],
         tv(1,[[],
            tv(0.9,[[],
               tv(0.9,[element(title,[],[Los trabajos de Persiles..]),...]),
   tv(0.8,[[],
     tv(0.9,[element(title,[],[La Celestina]),[],[]]),...
```

More details about our implementation of the flexible version of the XPath interpreter and debugger reported in this paper are available on: `http://dectau.uclm.es/fuzzyXPath/` (please, try with the on-line tools).

5 Conclusions and Future Work

In this paper we have presented an approach for XPath debugging. The result of the debugging process of a XPath expression is a set of alternative queries, each one associated to a chance degree. We have proposed JUMP, DELETE and SWAP operators that cover the main cases of programming errors when describing a path about a XML document. Our implemented and tested approach has a fuzzy taste in the sense that XPath expressions are debugged by relaxing the shape of path queries with chance degrees.

Although XML files are extensively used in many applications, the debugging of XPath queries has not been studied enough in the literature. Some authors have explored this topic [11], where the functional logic language TOY has been used for debugging XPath expressions. There the debugger is able to assist the programmer when a tag is wrong, providing alternative tags, and to trace executions. The current work can be considered as an extension of the quoted work. Here, our debugging technique gives to programmers a chance degree for each proposed alternative by annotating wrong-points on XPath expressions. We have based our approach in the works [12, 13], where XPath relaxation is studied by giving some rules for query rewriting: axis relaxation, step deletion and step cloning, among others. However, they do not give chance degrees associated to wrong XPath expressions.

We are nowadays introducing *thresholding* techniques on our fuzzy XPath in order to increase its performance when dealing with massive XML files. Our idea is to create filters for prematurely disregarding those superfluous computations dealing with non-significant solutions. In [14] we have reported some successful thresholding-based techniques specially tailored for MALP.

References

1. Berglund, A., Boag, S., Chamberlin, D., Fernandez, M., Kay, M., Robie, J., Siméon, J.: XML path language (XPath) 2.0. W3C (2007)
2. Almendros-Jiménez, J., Luna, A., Moreno, G.: A Flexible XPath-based Query Language Implemented with Fuzzy Logic Programming. In: Proc. of 5th Int. Symp. on Rules, RuleML'11, Springer Verlag, LNCS 6826 (2011) 186–193
3. Almendros-Jiménez, J., Luna, A., Moreno, G.: Fuzzy logic programming for implementing a flexible xpath-based query language. Electronic Notes in Theoretical Computer Science, Elsevier Science **282** (2012) 3–18
4. Almendros-Jiménez, J., Luna, A., Moreno, G.: A XPath Debugger based on Fuzzy Chance Degrees. In et al., P.H., ed.: Proc. of OTM'12 Workshops, Springer Verlag, LNCS 7567 (2012) 669–672
5. Lloyd, J.: Foundations of Logic Programming. Springer-Verlag, Berlin (1987)
6. Medina, J., Ojeda-Aciego, M., Vojtáš, P.: Similarity-based Unification: a multi-adjoint approach. Fuzzy Sets and Systems **146** (2004) 43–62
7. Morcillo, P., Moreno, G.: Programming with Fuzzy Logic Rules by using the FLOPER Tool. In et al., N.B., ed.: Proc. of 2nd. Int. Symp. on Rules, RuleML 2008, Springer Verlag, LNCS 3521 (2008) 119–126
8. Morcillo, P., Moreno, G., Penabad, J., Vázquez, C.: Fuzzy Computed Answers Collecting Proof Information. In et al., J.C., ed.: Advances in Computational Intelligence - Proc. of IWANN 2011, Springer Verlag, LNCS 6692 (2011) 445–452
9. Almendros-Jiménez, J.: An Encoding of XQuery in Prolog. In: Proc. of 6th Int. XML Database Symp. XSym'09, Springer, LNCS 5679 (2009) 145–155
10. Almendros-Jiménez, J., Becerra-Terón, A., Enciso-Baños, F.J.: Querying XML documents in logic programming. TPLP **8**(3) (2008) 323–361
11. Almendros-Jiménez, J., Caballero, R., GarcíA-Ruiz, Y., SáEnz-PéRez, F.: Xpath query processing in a functional-logic language. Electron. Notes Theor. Comput. Sci. **282** (May 2012) 19–34
12. Fazzinga, B., Flesca, S., Furfaro, F.: On the expressiveness of generalization rules for XPath query relaxation. In: Proceedings of the Fourteenth International Database Engineering & Applications Symposium, ACM (2010) 157–168
13. Fazzinga, B., Flesca, S., Furfaro, F.: Xpath query relaxation through rewriting rules. IEEE transactions on knowledge and data engineering **23**(10) (2011) 1583–1600
14. Julián, P., Medina, J., Morcillo, P., Moreno, G., Ojeda-Aciego, M.: A static preprocess for improving fuzzy thresholded tabulation. In et al., J.C., ed.: Advances in Computational Intelligence. Proc. of IWANN'11, Springer Verlag, LNCS 6692 (2011) 429–436