Electronic Communications of the EASST
Volume X (2012)

Proceedings of the XII Spanish Conference on Programming and Computer Languages (PROLE 2012)

Debugging Fuzzy XPath Queries

Jesús M. Almendros-Jiménez, Alejandro Luna and Ginés Moreno

15 pages

Guest Editors: María-del-Mar Gallardo
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Debugging Fuzzy XPath Queries

Jesús M. Almendros-Jiménez[1], Alejandro Luna[2] and Ginés Moreno[3]

[1] jalmen@ual.es
Dpto. de Lenguajes y Computación
Universidad de Almería
04120 Almería (Spain)
[2] Alejandro.Luna@alu.uclm.es
[3] Gines.Moreno@uclm.es
Dept. of Computing Systems
University of Castilla-La Mancha
02071 Albacete (Spain)

Abstract: In this paper we report a preliminary work about XPath debugging. We will describe how we can manipulate an XPath expression in order to obtain a set of alternative XPath expressions that match to a given XML document. For each alternative XPath expression we will give a chance degree that represents the degree in which the expression deviates from the initial expression. Thus, our work is focused on providing the programmer a repertoire of paths that (s)he can use to retrieve answers. The approach has been implemented and tested.

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 [BBC+07] 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 when (s)he describes the path in which the data are allocated. Tipically, (s)he omits some of the tags of the path, s(he) adds more than necessary, and (s)he 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 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 occur at several positions, and the programmer could find answers that do not correspond to her(is) 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 conditions that do not hold at the same time. When the programmer does not find the answer (s)he is looking for, there is a mechanism that (s)he 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, (s)he does not know whether they are close to h(er) 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.

In this paper we report a preliminary work about XPath debugging. We will describe how we can manipulate an XPath expression in order to obtain a set of alternative XPath expressions that match to a given XML document. For each alternative XPath expression we will give a chance degree that represents the degree in which the expression deviates from the initial expression. Thus, our work is focused on providing the programmer a repertoire of paths that (s)he can use to retrieve answers.

We propose that XPath debugging is guided by the programmer that initially establishes a value (a real value between 0 and 1), that the debugger uses to penalize each bug. Each bug found is penalized with such a value, and thus the chance degree is proportional to the value. Additionally, we assume that the debugger is equipped with a table of similarities, that is, a table in which pairs of similar words are assigned to a value between 0 and 1. It makes possible that chance degree is also computed from similarity degrees. The debugger reports a set of annotated paths in an extended XPath syntax in which we have incorporated 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, and `DELETE` is used to represent that a tag has been removed. The reported XPath expressions update the original XPath expression, that is, the case `JUMP` incorporates '//' at the position in which the bug is found, the case `SWAP` includes the new tag, and the case `DELETE` removes the wrong tag.

Additionally, our proposal permits the programmer tests the reported XPath expressions. The annotated XPath expressions can be executed obtaining a ranked set of answers with respect to the chance degree. It facilitates the process of debugging because the programmer can visualize the answers to each query.

The approach has been implemented and tested (see http://dectau.uclm.es/fuzzyXPath/debug_fuzzyXPathTest.php). The implementation has been developed on top of the recently proposed fuzzy XPath extension [ALM12], which uses *fuzzy logic programming* to provide a fuzzy taste to XPath expressions. Although our approach can be applied to standard (crisp) XPath expressions, chance degrees in XPath debugging fits well with our proposed framework. Particularly, XPath debugging annotations can be seen as annotations of XPath expressions similar to the proposed `DEEP` and `DOWN` of [ALM12]. `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. When annotated XPath expressions are executed, we obtain a ranked set of answers with respect to the *CD (Chance Degree)* of the programmer. DEEP and JUMP have, in fact, the same behavior: JUMP proportionally penalizes answers as deep as they occur. Finally, and in order to cover with SWAP, we have incorporated to our framework similarity degrees.

The structure of the paper is as follows. Section 2 will summarize our previous work and will introduce concepts that are closely related to the proposed debugging technique. Section 3 will describe the debugging technique. Section 4 will show some implementation details, and finally, Section 5 will conclude and present future work.

2 Fuzzy XPath

In this section we will summarize the main elements of our proposed fuzzy XPath language described in [ALM12, ALM11].

Our fuzzy XPath incorporates 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 document. Both structural constraints can be used together, assigning degree of importance 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, 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 number
deep :=	DEEP=number
down :=	DOWN=number
deep-down :=	deep | down | deep '**;**' down
num-op :=	**>** | **=** | **<** | **<>**
fuzzy-op :=	**and** | **and+** | **and-** | **or** | **or+** | **or-** | **avg** | **avg**{number,number}
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.

- **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

Figure 1: Fuzzy Logical Operators

$$
\begin{array}{lcllll}
\&_{\mathrm{P}}(x,y) & = & x*y & |_{\mathrm{P}}(x,y) & = & x+y-x*y \quad \textit{Product: and/or} \\
\&_{\mathrm{G}}(x,y) & = & \min(x,y) & |_{\mathrm{G}}(x,y) & = & \max(x,y) \quad \textit{Gödel: and+/or-} \\
\&_{\mathrm{L}}(x,y) & = & \max(x+y-1,0) & |_{\mathrm{L}}(x,y) & = & \min(x+y,1) \quad \textit{Łuka.: and-/or+}
\end{array}
$$

Figure 2: XML skeleton represented as a tree

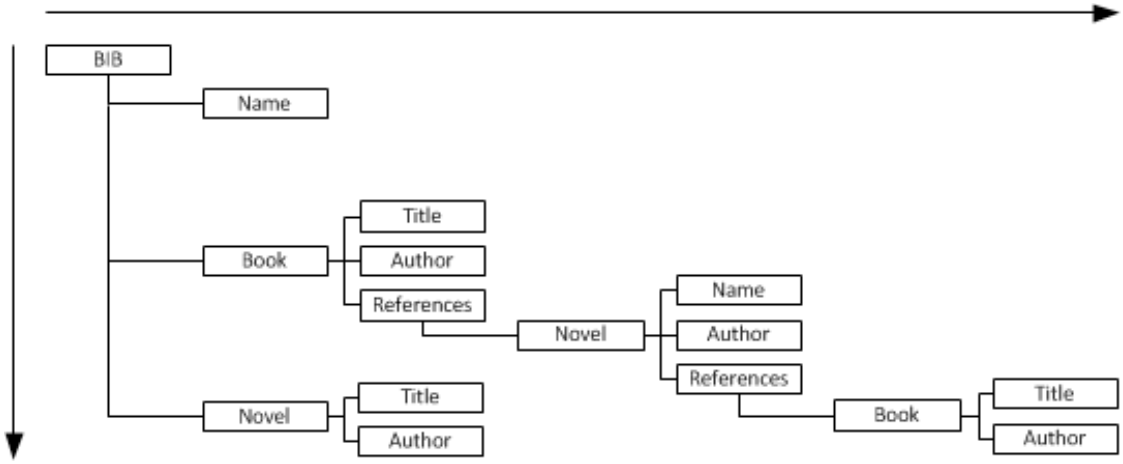

Figure 3: Input XML document in our examples

```
<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>
```

illustrate these explanations, let us see some examples of our proposed fuzzy version of XPath according to the XML document shown of Figure 3, whose *skeleton* is depicted in Figure 2.

Figure 4: 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> <title **rsv**="0.7200">La Celestina</title> <title **rsv**="0.2949">Los trabajos de Persiles y Sigismunda</title> </result>	$0.8000 = 0.8$ $0.7200 = 0.8 * 0.9$ $0.2949 = 0.8^5 * 0.9$

Figure 5: 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 Sigismunda</title> <title **rsv**="0.25">Don Quijote de la Mancha</title> </result>	$1.00 = (3*1+1*1)/(3+1)$ $0.25 = (3*0+1*1)/(3+1)$

Figure 6: 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> <title **rsv**="0.0625">Los trabajos de Persiles y ..</title> </result>	$0.25 = (3*0+1*1)/(3+1)$ $0.0625 = 0.5^4 * (3*1+1*1)/(3+1)$

Example 1 *Let us consider the fuzzy XPath query of Figure 4 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 in Figure 4. 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 5 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 6 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 $[\texttt{DEBUG} = r]$ 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 Figures 2 and 3, the debugging produces a set of alternative queries $Q_1, ..., Q_n$ packed into an output XML document, like the one shown in Figure 7. The document

has the following structure:

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

where the set of alternatives 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.

Figure 7: Debugging of the 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>
```

In Figure 7, the first alternative, with the highest CD, is just the original query, thus, the CD is *1*, whose further execution with fuzzy XPath returns «Don Quijote de La Mancha». As was commented in the introduction, we have assumed the debugger is ran 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.

In order to explain the way in which our technique generates the attributes and content of each *query* tag in the output XML document, let us consider a generic path Q of the form: «$[\texttt{DEBUG} = r]/tag_1/.../tag_i/tag_{i+1}/...$», where we say that tag_i is at level i in the original query. So, assume that during the exploration of the input query Q and the 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:

Swapping case: Instead of tag_i, we find tag'_i at level i in the input XML document D, being tag_i and tag'_i 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 "$[\texttt{SWAP} = s]tag'_i/$".
The second query proposed in Figure 7 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, we can run the (fuzzy) XPath queries «/bib/novel/title» and even «/bib/[SWAP=0.8]novel-/title» (see Figure 8). In both cases we obtain the same result, i.e., «La Celestina», but with different RSVs (1 and 0.8).

Figure 8: Execution of the query «/bib/[SWAP=0.8]novel/title»

```
<result>
  <title rsv="0.8">La Celestina</title>
</result>
```

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]//"*, being r the value associated to DEBUG.

Figure 9: Execution of the query «/bib/[JUMP=0.5]//title»

```
<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>
```

Figure 10: Execution of the query «/[JUMP=0.5]//book/title»

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

This situation is illustrated by the third and fourth queries in Figure 7, where we propose to jump the tags *book* and *bib*. The execution of the queries returns different results, as shown in

Figures 9 and 10, where JUMP produces similar effects to the DEEP command explained in the previous section, that is, as more tags are jumped their resulting CD's become lower.

Deletion case: This situation 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 7, 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.

As seen in Figure 11, the execution of our new query is able to retrieve the information contained in the first branch of the input XML document listed in Figures 2 and 3. Here we illustrate that execution of debugged XPath expressions reveals hidden answers that can fulfill the programmer expectations.

Figure 11: Execution of the query «/bib/[DELETE=0.5][SWAP=0.9]name»

```
<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.

In Figure 12, we can see the execution of the query:

« <query cd="0.225" bib="" book="//" title="name">/[DELETE=0.5][JUMP=0.5]// [SWAP=0.9]name</query> »

The CD *0.225* is quite low, and therefore the chance degree is low, 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*).

Figure 12: Execution of the query «/[DELETE=0.5][JUMP=0.5]//[SWAP=0.9]name»

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

The wide range of alternatives (Figure 7 is still incomplete), reveals the flexibility of our technique. The programmer is free to use the alternative queries to execute them, and to inspect results up to the expectations are covered.

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 13 (compare it with Figure 7) we show the result of debugging the following query: «[DEBUG=0.7]/bib/[DEBUG=0.6]book/[DEBUG=0.5]title».

Figure 13: 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>
```

4 Implementation based on Fuzzy Logic Programming with MALP

Multi-Adjoint Logic Programming [MOV04], MALP in brief, is based on a first order language, $\mathscr{L}$, containing variables, function/constant symbols, predicate symbols, and several arbitrary connectives such as implications $(\leftarrow_1, \leftarrow_2, \ldots, \leftarrow_m)$, conjunctions $(\&_1, \&_2, \ldots, \&_k)$, disjunctions $(\vee_1, \vee_2, \ldots, \vee_l)$, and general hybrid operators ("aggregators" $@_1, @_2, \ldots, @_n$), used for combining/propagating truth values through the rules, and thus increasing the language expressiveness. Additionally, our language $\mathscr{L}$ contains the values of a *multi-adjoint lattice* in the form $\langle L, \preceq, \leftarrow_1, \&_1, \ldots, \leftarrow_n, \&_n \rangle$, equipped with a collection of *adjoint pairs* $\langle \leftarrow_i, \&_i \rangle$ where each $\&_i$ is a conjunctor intended to the evaluation of *modus ponens*. A *rule* is a formula "$A \leftarrow_i \mathscr{B}$ with α", where A is an atomic formula (usually called the *head*), $\mathscr{B}$ (which is called the *body*) is a formula built from atomic formulas $B_1, \ldots, B_n$ $(n \geq 0)$, truth values of L and conjunctions, disjunctions and general aggregations, and finally $\alpha \in L$ is the "weight" or *truth degree* of the rule. The set of truth values L may be the carrier of any complete bounded lattice, as for instance occurs with the set of real numbers in the interval $[0,1]$ with their corresponding ordering $\preceq_R$. Consider, for instance, the following program composed by three rules with associated multi-adjoint lattice $\langle [0,1], \preceq_R, \leftarrow_{\mathrm{P}}, \&_{\mathrm{P}} \rangle$ (where label P means for *Product logic* with the following connective definitions for implication, conjunction and disjunction symbols, respectively: "$\leftarrow_{\mathrm{P}}(x,y) = \min(1, x/y)$", "$\&_{\mathrm{P}}(x,y) = x * y$" and "$|_{\mathrm{P}}(x,y) = x + y - x * y$"):

$$\begin{array}{lllll} \mathscr{R}_1: & p(X) & \leftarrow_{\mathtt{P}} & q(X,Y) \mid_{\mathtt{P}} r(Y) & with \quad 0.8 \\ \mathscr{R}_2: & q(a,Y) & \leftarrow & & with \quad 0.9 \\ \mathscr{R}_3: & r(b) & \leftarrow & & with \quad 0.7 \end{array}$$

In order to run and manage MALP programs, during the last years we have designed the FLOPER system [MM08, MMPV10, MMPV11], which is freely accessible from the Web site http://dectau.uclm.es/floper/. The parser of our tool has been implemented by using the classical DCG's (*Definite Clause Grammars*) resource of the Prolog language, since it is a convenient notation for expressing grammar rules. Once the application is loaded inside a Prolog interpreter, it shows a menu which includes options for loading/compiling, parsing, listing and saving fuzzy programs, as well as for executing/debugging fuzzy goals. All these actions are based on the *compilation* of the fuzzy code into standard Prolog code.

The key point of the compilation is to extend each atom with an extra argument, called *truth variable* of the form "`_TV`$_i$", which is intended to contain the truth degree obtained after the subsequent evaluation of the atom. For instance, the first fuzzy rule of our previous program is translated to clause:

```
p(X,_TV0):-q(X,Y,_TV1),r(Y,_TV2),or_prod(_TV1,_TV2,_TV3),and_prod(0.8,_TV3,_TV0).
```

Moreover, the remaining rules become the pure Prolog facts "`q(a,Y,0.9)`" and "`r(b,0.7)`", whereas the corresponding lattice is expressed by the following Prolog clauses (where the meaning of the mandatory predicates `member`, `top`, `bot` and `leq` is obvious):

```
member(X):- number(X), 0=<X, X=<1.                          bot(0).
leq(X,Y):- X=<Y.                                           top(1).
and_prod(X,Y,Z):- pri_prod(X,Y,Z).    pri_prod(X,Y,Z):- Z is X * Y.
or_prod(X,Y,Z):- pri_prod(X,Y,U1),pri_add(X,Y,U2),pri_sub(U2,U1,Z).
pri_add(X,Y,Z):- Z is X+Y.                pri_sub(X,Y,Z):- Z is X-Y.
```

Finally, a fuzzy goal like "`p(X)`", is translated into the pure Prolog goal: "`p(X, Truth_degree)`" (note that the last truth degree variable is not anonymous now) for which, after choosing in FLOPER option "`run`", the underlying Prolog interpreter computes the desired fuzzy answer `[Truth_degree=0.776,X=a]`. Note that all internal computations (including compiling and executing) are pure Prolog derivations, whereas inputs (fuzzy programs and goals) and outputs (fuzzy computed answers) have always a fuzzy taste, thus producing the illusion on the final user of being working with a purely fuzzy logic programming tool. By using option "`lat`" ("`show`") of FLOPER, we can associate (and display) a new lattice to a given program. As seen before, such lattices must be expressed by means of a set of Prolog clauses (defining predicates `member`, `top`, `bot`, `leq` and the ones associated to fuzzy connectives) in order to be loaded into FLOPER.

4.1 MALP and XPath

MALP can be used as basis for our proposed fuzzy extension of XPath as follows. The idea is to implement XPath by means of MALP rules.

Firstly, we make use of a SWI-Prolog library for loading XML files in order to store each XML

document by means of a Prolog term[1] representing a tree. 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 list containing the attributes, and `Subelements` is a list containing the sub-elements (i.e. subtrees) of the tag. For instance, the XML document of Figure 3 is represented in SWI-Prolog as:

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

Secondly, for loading XML documents in our implementation we make use of the SWI-Prolog predicate `load_xml(+File,-Term)`. Similarly, we have a predicate `write_xml(+File,+Term)` for writing a data-term representing an XML document into a file.

Thirdly, we have to consider a complete lattice L_{tv} to be used for representing the RSV's and the CD's. L_{tv} contains trees, which we will call *tv trees*, of the form: "$tv(v,[root,tvch,tvsib])$", where v is a value taken from $[0,1]$ (i.e., the RSV or CD), *root* is the root of the tree to which v is associated, and *tvch*, *tvsib* are the *tv trees* of the children and sibling nodes, respectively.

Finally, MALP rules have the form "$A \leftarrow \mathscr{B}$ *with tvtree*". In addition, some XPath fuzzy operators *and*, *or* and *avg* can be mapped to MALP connectives in lattice L_{tv}. Finally, we have to consider an auxiliary aggregator in L_{tv} called @*fuse*, which builds the *tv tree* of a certain node from the tv trees of the sibling and children nodes. For instance, some representative Prolog clauses (used in FLOPER) defining the new lattice based on *tv trees* are:

```
member(tv(N,L)):-number(N),0=<N,N=<1,(L=[];L=[_|_]).          bot(tv(0,[])).
and_prod(tv(X1,X2),tv(Y1,Y2),tv(Z1,Z2)):-pri_prod(X1,Y1,Z1),pri_app(X2,Y2,Z2).
pri_app([],X,X).                    pri_app([A|B],C,[A|D]):-pri_app(B,C,D).
```

4.2 MALP and the XPath Debugger

The core of our debugger is coded with MALP rules by reusing most modules of our fuzzy XPath interpreter [ALM11, ALM12]. And, of course, the parser of our debugger has been extended to recognize the new keywords `DEBUG`, `SWAP`, `DELETE` and `JUMP`, with their proper arguments.

Now, we would like to show how the new "*XPath debugging*" predicate admits an elegant definition by means of fuzzy MALP rules. Each rule defining predicate:

$$\texttt{debugQuery}(\texttt{ListXPath},\texttt{Tree},\texttt{Penalty})$$

receives three arguments: (1) `ListXPath` is the Prolog representation of an XPath expression, (2) `Tree` is the term representing an input XML document and (3) `Penalty` represents the *chance degree*. A call to this predicate returns a truth-value (i.e., a *tv tree*) like the following one:

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

```
tv(1.0,[[/,[]],
   tv(1.0, [[tag(bib),[]],
      tv(1.0,[[tag(book),[]],
         tv(1.0,[[tag(title),[]],[],
         ...
      tv(0.8,[[tag(novel),[book=novel]],
      ...
      tv(0.5,[['[DELETE=0.5]',[book='']],
         tv(0.9,[[tag(name),[title=name]],[],
         ...
         [])]),
      [])]),
   [])])])]),
[])
```

Basically, the *debugQuery* predicate traverses the XML document, checking the validity of the original query *tag-by-tag*, and trying to match each *tag* to the ones occurring in the XML document and thus producing effects of `DELETE`, `JUMP` and `SWAP`.

The definition of such predicate includes several rules for distinguishing the three debugging cases. As an example the case of swapping is defined as follows:

```
debugQuery([Label|LabelRest],[element(Label2,_,Children)|Siblings],Penalty) <prod
       @fuse(
               similarity(Label, Label2),
               debugQuery(LabelRest, Children, Penalty),
               debugQuery([Label|LabelRest], Siblings, Penalty)
       ) with tv(1,[]).
```

which becomes into the following Prolog clause after compilation into FLOPER:

```
debugQuery([Label|LabelRest],[element(Label2,_,Children)|Siblings], Penalty, TV_Iam):-
       similarity(Label, Label2, TV_Similarity),
       debugQuery(LabelRest, Children, Penalty, TV_Son),
       debugQuery([Label|LabelRest], Siblings, Penalty, TV_Sib),
       agr_fuse(TV_Similarity, TV_Son, TV_Sib, TV_Iam).
```

Basically, similarity is checked with the atom `similarity(Label, Label2)`, and two recursive calls are achieved for debugging both children (`debugQuery(LabelRest, Children, Penalty)`) and siblings (`debugQuery([Label|LabelRest], Siblings, Penalty, 1)`), whose *tv trees* are finally combined (*fused*) with the node content.

Finally, for showing the result in a pretty way (see Figures 7), and transforming a *tv tree* into an XML file, a predicate *tv_to_element* has been implemented.

5 Conclusions and Future Work

In this paper we have presented a proposal of 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 an XML document. We have implemented and tested the approach. The approach has a fuzzy taste in the sense that XPath expressions are debugged by relaxing the path expression assigning a chance degree to debugging points. The fuzzy taste is also illustrated when executing the annotated XPath expressions in a fuzzy based implementation of XPath.

Although XML and XPath are extensively used in many applications, the debugging of XPath has not been explored enough in the literature. Some authors have explored this topic [ACGS12], 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, the debugging gives to programmers a chance degree for each tag alternative, and annotates XPath expressions in the points in which the error was found. We have based our work in the proposal on [FFF10, FFF11] where XPath relaxation is studied given some rules for query rewriting: axis relaxation, step deletion and step cloning, among others. However, they do not give chance degree associated to the input XPath expression.

Our future work will focus on incorporating new elements to the debugging. The first one is the handling of boolean conditions. Boolean conditions can express ranges that can be wrong and conditions that cannot be satisfied. The second one is to provide more flexibility to the debugger language: `DEBUG` is used in the debugger to annotate the penalization, and it is used for `JUMP`, `DELETE` and `SWAP`. A more flexible case would be to provide a different penalization in each case.

Acknowledgements: This work has been partially supported by the EU, under FEDER, and the Spanish Science and Innovation Ministry (MICINN) under grants TIN2008-06622-C03-03 and TIN2007-65749, as well as by Ingenieros Alborada IDI under grant TRA2009-0309, the Castilla-La Mancha Administration under grant PII1I09-0117-4481, and the JUNTA ANDALUCIA administration under grant TIC-6114 (proyecto de excelencia).

Bibliography

[ACGS12] J. M. Almendros-Jiménez, R. Caballero, Y. GarcíA-Ruiz, F. SáEnz-PéRez. XPath Query Processing in a Functional-Logic Language. *Electron. Notes Theor. Comput. Sci.* 282:19–34, May 2012.

[ALM11] J. Almendros-Jiménez, A. Luna, G. Moreno. A Flexible XPath-based Query Language Implemented with Fuzzy Logic Programming. In *Proc. of 5th International Symposium on Rules: Research Based, Industry Focused, RuleML'11. Barcelona, Spain, July 19–21*. Pp. 186–193. Springer Verlag, LNCS 6826, Heidelberg,Germany, 2011.

[ALM12] J. M. Almendros-Jiménez, A. Luna, G. Moreno. Fuzzy Logic Programming for Implementing a Flexible XPath-based Query Language. *Electr. Notes Theor. Comput. Sci.* 282:3–18, 2012.

[BBC+07] A. Berglund, S. Boag, D. Chamberlin, M. Fernandez, M. Kay, J. Robie, J. Siméon. XML path language (XPath) 2.0. *W3C*, 2007.

[FFF10] B. Fazzinga, S. Flesca, F. Furfaro. On the expressiveness of generalization rules for XPath query relaxation. In *Proceedings of the Fourteenth International Database Engineering & Applications Symposium*. Pp. 157–168. 2010.

[FFF11] B. Fazzinga, S. Flesca, F. Furfaro. XPath Query Relaxation through Rewriting Rules. *IEEE transactions on knowledge and data engineering* 23(10):1583–1600, 2011.

[MM08] P. Morcillo, G. Moreno. Programming with Fuzzy Logic Rules by using the FLOPER Tool. In al. (ed.), *Proc of the 2nd. Rule Representation, Interchange and Reasoning on the Web, International Symposium, RuleML'08*. Pp. 119–126. Springer Verlag, LNCS 3521, 2008.

[MMPV10] P. Morcillo, G. Moreno, J. Penabad, C. Vázquez. A Practical Management of Fuzzy Truth Degrees using FLOPER. In al. (ed.), *Proc. of 4nd Intl Symposium on Rule Interchange and Applications, RuleML'10*. Pp. 20–34. Springer Verlag, LNCS 6403, 2010.

[MMPV11] P. Morcillo, G. Moreno, J. Penabad, C. Vázquez. Fuzzy Computed Answers Collecting Proof Information. In al. (ed.), *Advances in Computational Intelligence - Proc of the 11th International Work-Conference on Artificial Neural Networks, IWANN 2011*. Pp. 445–452. Springer Verlag, LNCS 6692, 2011.

[MOV04] J. Medina, M. Ojeda-Aciego, P. Vojtáš. Similarity-based Unification: a multi-adjoint approach. *Fuzzy Sets and Systems* 146:43–62, 2004.