Analyzing Fuzzy Logic Computations with Fuzzy XPath

Jesús M. Almendros-Jiménez[1], Alejandro Luna[2], Ginés Moreno[3] and Carlos Vázquez [4]

[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
[4] Carlos.Vazquez@uclm.es
Dept. of Computing Systems
University of Castilla-La Mancha
02071 Albacete (Spain)

Abstract: Implemented with a fuzzy logic language by using the FLOPER tool developed in our research group, we have recently designed a fuzzy dialect of the popular XPath language for the flexible manipulation of XML documents. In this paper we focus on the ability of *Fuzzy XPath* for exploring derivation trees generated by FLOPER once they are exported in XML format, which somehow serves as a debugging/analizing tool for discovering the set of fuzzy computed answers for a given goal, performing depth/breadth-first traversals of its associated derivation tree, finding non fully evaluated branches, etc., thus reinforcing the bi-lateral synergies between *Fuzzy XPath* and FLOPER.

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

1 Introduction

Logic Programming (LP) [Llo87] is being widely used from several decades ago for problem solving and knowledge representation, thus providing a great amount of foundations and techniques devoted to produce real world applications. Some steps beyond, during the last years important research efforts have been performed for introducing inside the LP paradigm some techniques/constructs based on fuzzy logic in order to explicitly treat with uncertainty and approximated reasoning in a natural way. Following this trail, several fuzzy logic programming systems have been developed [KS92, BMP95, Voj01, GMV04, MCS11], where the classical inference mechanism of SLD–Resolution has been replaced by a fuzzy variant which is able to handle partial truth in a comfortable way.

This is the case too of *Multi-Adjoint Logic Programming* [MOV04], MALP in brief, where a fuzzy program can be seen as a set of rules each one annotated with its own truth degree (a value of a complete lattice, for instance, the real interval $[0,1]$). Goals are evaluated in two separate computational phases. During the *operational* phase, *admissible steps* (a generalization of the

classical *modus ponens* inference rule) are systematically applied by a backward reasoning procedure in a similar way to classical resolution steps in pure logic programming. More precisely, in an admissible step, for a selected atom A in a goal and a rule $\langle H \leftarrow \mathscr{B}; v \rangle$ of the program, if there is a most general unifier θ of A and H, then atom A is substituted by the expression $(v \& \mathscr{B})\theta$, where "&" is an adjoint conjunction evaluating *modus ponens*. Finally, the operational phase returns a computed substitution together with an expression where all atoms have been exploited. This last expression is then interpreted under a given lattice during what we call the *interpretive* phase, hence returning a pair $\langle$*truth degree*; *substitution*$\rangle$ which is the fuzzy counterpart of the classical notion of computed answer traditionally used in pure logic programming.

On the other hand, 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 [ALM11, ALM12a] we have presented an XPath interpreter (together with a debugger, as documented in [ALM12b, ALM13]) extended with fuzzy commands which somehow rely on the implementation based on fuzzy logic programming by using FLOPER.

Whereas in Sections 2 and 3 we summarize the main features of both the *fuzzy XPath* interpreter and the fuzzy logic programming environment FLOPER, respectively, in Section 4 we go deeper on the feedbacks between both tools. More exactly we show that, even when FLOPER was used for implementing *fuzzy XPath*, now this last language is very useful for formulating queries to be executed againts XML documents representing derivation trees depicted by FLOPER, thus becoming into a "debugging" technique which can be embedded into the programming environment for analyzing some interesting details (fuzzy computed answers, tree traversals, partial branches, etc.) about fuzzy logic computations. Finally, in Section 5 we 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 [ALM12a, ALM11] (the tool can be freely downloaded and tested on-line in `http://dectau.uclm.es/fuzzyXPath/`). On this flexible dialect of XPath, we have incorporated 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 con-

ditions, 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:

Figure 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+*

– **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 (t-norms and t-conorms, respectively [SS83, KMP00]), 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 (*Retrieval Status Values*) 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. $avg\{p_1,p_2\}$, acts as $(p_1*r_1+p_2*r_2)/(p_1+p_2)$.

Figure 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>
```

Figure 3: Execution of 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...</title> </result>	$0.8000 = 0.8$ $0.7200 = 0.8 * 0.9$ $0.2949 = 0.8^5 * 0.9$

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

Document	RSV computation
<result> <title **rsv**="1.00">Los trabajos de Persiles...</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 5: Execution of 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...</title> </result>	$0.25 = (3*0+1*1)/(3+1)$ $0.0625 = 0.5^4 * (3*1+1*1)/(3+1)$

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 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.*

Finally, we can use command «[FILTER = *r*]» at the beginning of a query for filtering its final set of solutions in the sense that only those ones with RSV not lower than *r* will conform the output.

3 Fuzzy Logic Programming with MALP and FLOPER

Multi-Adjoint Logic Programming [MOV04], MALP in brief, can be thought as a fuzzy extension of Prolog and it 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* [SS83, KMP00, MOV04]. 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, $\mathscr{P}$, with associated multi-adjoint lattice $\langle [0,1], \preceq_R, \leftarrow_{\mathtt{P}}, \&_{\mathtt{P}} \rangle$ (where label P means for *Product logic* with the following connective definitions for implication and conjunction symbols, respectively: "$\leftarrow_{\mathtt{P}}(x,y) = \min(1, x/y)$", "$\&_{\mathtt{P}}(x,y) = x * y$", as well as "$@_{\mathtt{aver}}(x,y) = (x+y)/2$"):

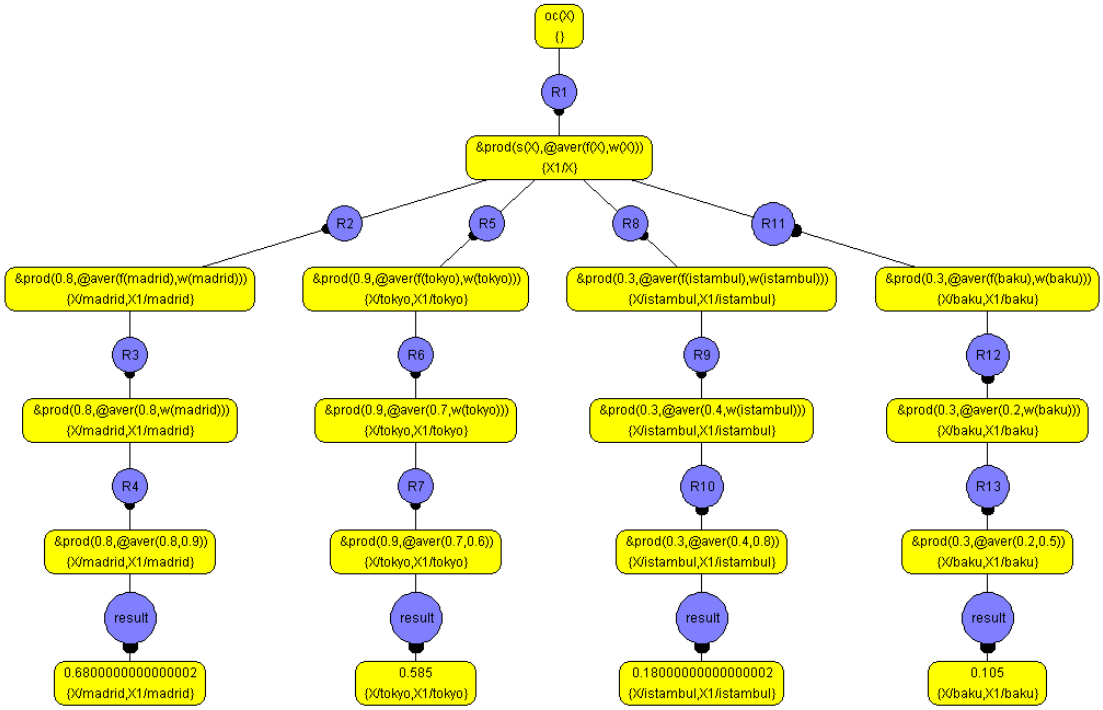
$$
\begin{array}{llllll}
\mathscr{R}_1: & oc(X) \text{ <- } s(X)\ \&prod\ (f(X) & @aver\ w(X)) & with\ 1. \\
\mathscr{R}_2: & s(madrid) & with\ 0.8. & \mathscr{R}_5: & s(tokyo) & with\ 0.9. \\
\mathscr{R}_3: & f(madrid) & with\ 0.8. & \mathscr{R}_6: & f(tokyo) & with\ 0.7. \\
\mathscr{R}_4: & w(madrid) & with\ 0.9. & \mathscr{R}_7: & w(tokyo) & with\ 0.6. \\
\mathscr{R}_8: & s(istambul) & with\ 0.3. & \mathscr{R}_{11}: & s(baku) & with\ 0.3. \\
\mathscr{R}_9: & f(istambul) & with\ 0.4. & \mathscr{R}_{12}: & f(baku) & with\ 0.2. \\
\mathscr{R}_{10}: & w(istambul) & with\ 0.8. & \mathscr{R}_{13}: & w(baku) & with\ 0.5.
\end{array}
$$

This program models, through predicate `oc`/1, the chances of a city for being an "olympic city" (i.e., for hosting olympic games). Predicate `oc`/1 is defined in rule $\mathscr{R}_1$, whose body collects the information from three other predicates, `s`/1, `f`/1 and `w`/1, modeling, respectively, the *s*ecurity level, the *f*acilities and the good *w*eather of a certain city. These predicates are defined in rules $\mathscr{R}_2$ to $\mathscr{R}_{13}$ for four cities (*Madrid*, *Istambul*, *Tokyo* and *Baku*), in such a way that, for each city, the characteristic modeled by each predicate is better the greater the truth value of the rule.

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 FLOPER system is able to manage programs with very different lattices. In order to associate a certain lattice with its corresponding program, such lattice must be loaded into the

Figure 6: Execution tree for program $\mathscr{P}$ and goal `oc(X)`



tool as a pure Prolog program. As an example, the following clauses show the program modeling the lattice of the real interval $[0,1]$ with the usual ordering relation and connectives (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.
```

FLOPER includes two main ways for evaluating a goal, given a MALP program and its corresponding lattice. Option "`run`" translates the whole program into a pure Prolog program and evaluates the (also translated) goal, thus obtaining a set of fuzzy computed answers, whereas, on the other hand, option "`tree`" displays the execution (or derivation) tree for the intended goal[1]. For the purpose of this paper, we will focus on this last option in order to obtain a tree (detailing the whole computational behaviour) for being afterwards analyzed with fuzzy XPath.

[1] Users can select the deepest level to be built, which is obviously mandatory when trees are infinite.

Let us consider the previously described program $\mathscr{P}$, and goal "`oc(X)`", that asks for the eligibility of each one of the four cities in $\mathscr{P}$ as "Olympic City". We use option "`tree`" to obtain the execution tree, which is generated by FLOPER in three different formats. Firstly the tree is displayed in graphical mode, as a PNG file, as shown in Figure 6. The tree is composed by two kinds of nodes. Yellow nodes represent states reached by FLOPER following the state transition system that describes the operational semantics of MALP [MOV04]. The up-most node represents the first state (that is, the goal and the identity substitution), and subsequent lower nodes are its children states (that is, states reached from the goal). A state contains a formula in the upper side and a substitution (the record of substitutions applied from the original goal to reach that state) at the bottom. A final state, if reached, is a fuzzy computed answer, that is, its formula is an element of the lattice. Blue rounded nodes that intermediate between a pair of yellow nodes (a pair of states) represent program rules; specifically, the program rule that is exploited in order to go from one state (the upper state) to another (the lower one). These rules are named with letter "`R`" plus its position in the program. For example, observe that from the initial state to the next state, rule $\mathscr{R}_1$ of the program has been exploited, as shown in the blue intermediate node. As an exception, when all atoms have been exploited in (the formula of) a certain state, the following blue node indicates "`result`", informing that the next state is a fuzzy computed answer.

FLOPER can also generate the execution tree in two textual formats. The first one contains a plain description of the tree, while the second one provides an XML structure to that description, therefore becoming the focus of interest of this paper. In this XML format we define the tag "`node`" to contain all the information of a node, such as the rule performed to reach that state (that is "`R0`" in the case of the first state), the formula of the state, the accumulated substitution and the children nodes, given by the tags "`rule`", "`goal`", "`substitution`" and "`children`", respectively. The content of tags "`rule`", "`goal`" and "`substitution`" is a string, while the content of the tag "`children`" is a set of tags "`node`", as seen in the following lines, corresponding to the XML file associated to the the tree depicted in Figure 6.

```
<node>
    <rule>R0</rule>
    <goal>oc(X)</goal>
    <substitution>{}</substitution>
    <children>
        <node>
            <rule>R1</rule>
            <goal>and_prod(s(X),agr_aver(f(X),w(X)))</goal>
            <substitution>{X1/X}</substitution>
            <children>
                <node>
                    <rule>R2</rule>
                    <goal>and_prod(0.8,agr_aver(f(madrid),w(madrid)))</goal>
                    <substitution>{X/madrid,X1/madrid}</substitution>
                    <children>
                        <node>
                            <rule>R3</rule>
                            <goal>and_prod(0.8,agr_aver(0.8,w(madrid)))</goal>
                            <substitution>{X/madrid,X1/madrid}</substitution>
                            <children>
                                <node>
                                    <rule>R4</rule>
                                    <goal>and_prod(0.8,agr_aver(0.8,0.9))</goal>
```

```
                                    <substitution>{X/madrid,X1/madrid}</subtitution>
                                    <children>
                                        <node>
                                            <rule>result</rule>
                                            <goal>0.6800000000000002</goal>
                                            <substitution>{X/madrid,X1/madrid}
                                                </substitution>
                                            <children>
                                            </children>
                                        </node>
                                    </children>
                                </node>
                            </children>
                        </node>
                    </children>
                </node>
                ...
                                    <node>
                                        <rule>result</rule>
                                        <goal>0.585</goal>
                                        <substitution>{X/tokyo,X1/tokyo}
                                            </substitution>
                                        <children>
                                        </children>
                                    </node>
                ...
            </node>
        </children>
</node>
```

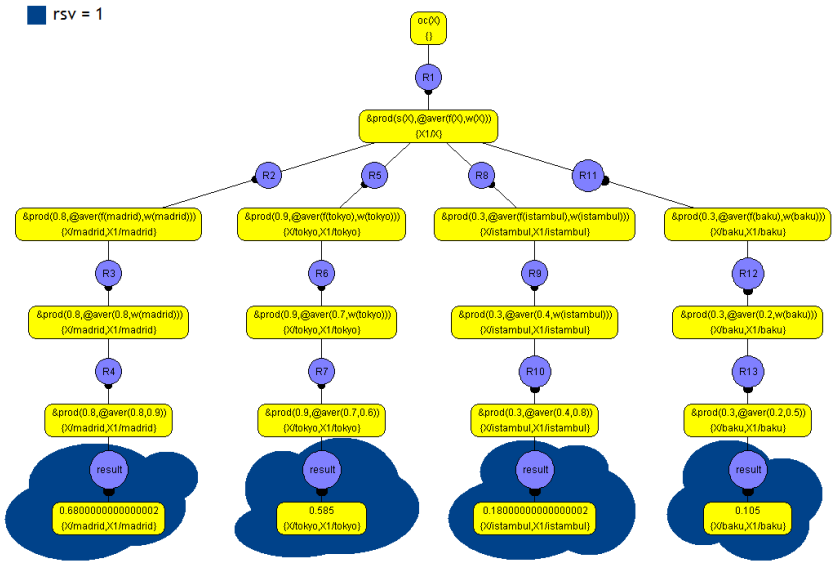
4 Exploring Derivation Trees with Fuzzy XPath

In this section we present a very powerful method to automatically exploring the behaviour of a MALP program using the *fuzzy XPath* tool described in Section 2. The idea is to use *fuzzy XPath* over the execution tree generated by FLOPER for a certain program and goal. That tree is obtained through option "`tree`" using the XML format just explained before in Section 3. For instance, an easy but interesting XPath query should be "`//node/rule`" which lists all the rules exploited along the execution of a goal (in the case of the tree depicted in Figure 6, we would obtain the whole set of rules defined in the program $\mathscr{P}$ of our running example).

Assume now that we plan to obtain the whole set of fuzzy computed answers for a given goal and program. This information, always collected in the leaves of execution trees (even when there exists the possibility of finding leaves non containing fuzzy computed answers, as we will see afterwards) as illustrated in Figure 7, can be retrieved by means of the *fuzzy XPath* query "`//node[/rule/text()=result]`", meaning that, return each *node* such that the content of its *rule* tag is "result". The XML text shown below Figure 7 represents the output of our *fuzzy XPath* interpreter for that query, where the selected nodes have been highlighted inside a blue cloud into the drawn tree above. Note that the resuting XML file contains four solutions (one for each city), where attribute "`rsv`" indicates how much each city fulfills the original query (in this example, this value is the same in all cases, that is, just the maximum one 1).

Strongly related with the previous experiment, but not directly focusing now on fuzzy computed answers, query "`//node[children[not(text())]]`" returns the leaves of the tree.

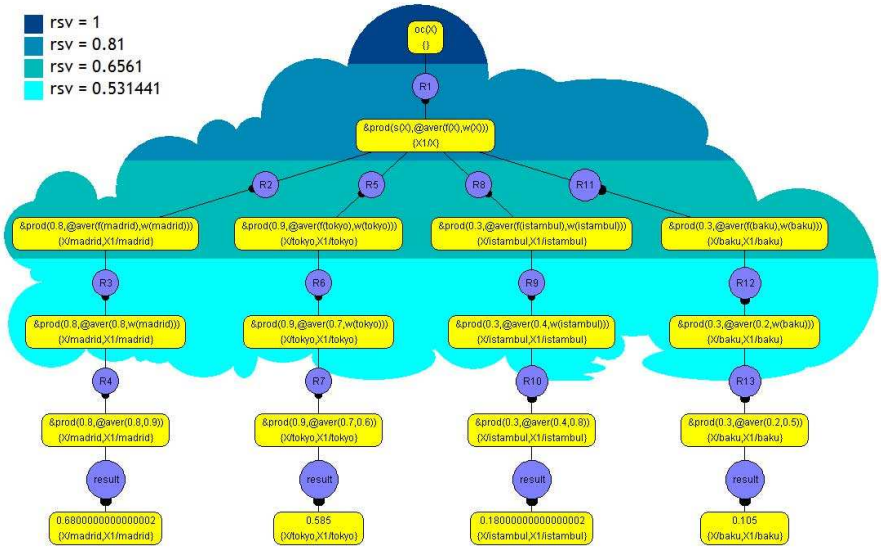
Figure 7: Executing queries «//node[/rule/text()=result]» and «//node[children[not(text())]]»




```
<result>
  <node rsv="1.0">
    <rule>result</rule>
    <goal>0.6800000000000002</goal>
    <substitution>{X/madrid, X1/madrid}</substitution>
    <children></children>
  </node>
  <node rsv="1.0">
    <rule>result</rule>
    <goal>0.585</goal>
    <substitution>{X/tokyo, X1/tokyo}</substitution>
    <children></children>
  </node>
  <node rsv="1.0">
    <rule>result</rule>
    <goal>0.18000000000000002</goal>
    <substitution>{X/istambul, X1/istambul}</substitution>
    <children></children>
  </node>
  <node rsv="1.0">
    <rule>result</rule>
    <goal>0.105</goal>
    <substitution>{X/baku, X1/baku}</substitution>
    <children></children>
  </node>
</result>
```

Note that, in the case of our current program $\mathscr{P}$ and goal "`oc(X)`", the corresponding output for this query is, once again, the same than the one reported previously in Figure 7 but, as said in the previous paragraph, this is not the general case. In fact, we can formulate a query like

Figure 8: Executing query «[FILTER=0.5][DEEP=0.9]//node/goal»




```
<result>
  <goal rsv="1.0">oc(X)</goal>
  <goal rsv="0.81">and_prod(s(X),agr_aver(f(X),w(X)))</goal>
  <goal rsv="0.6561">and_prod(0.8,agr_aver(f(madrid),w(madrid)))</goal>
  <goal rsv="0.6561">and_prod(0.9,agr_aver(f(tokyo),w(tokyo)))</goal>
  <goal rsv="0.6561">and_prod(0.3,agr_aver(f(istambul),w(istambul)))</goal>
  <goal rsv="0.6561">and_prod(0.3,agr_aver(f(baku),w(baku)))</goal>
  <goal rsv="0.531441">and_prod(0.8,agr_aver(0.8,w(madrid)))</goal>
  <goal rsv="0.531441">and_prod(0.9,agr_aver(0.7,w(tokyo)))</goal>
  <goal rsv="0.531441">and_prod(0.3,agr_aver(0.4,w(istambul)))</goal>
  <goal rsv="0.531441">and_prod(0.3,agr_aver(0.2,w(baku)))</goal>
</result>
```

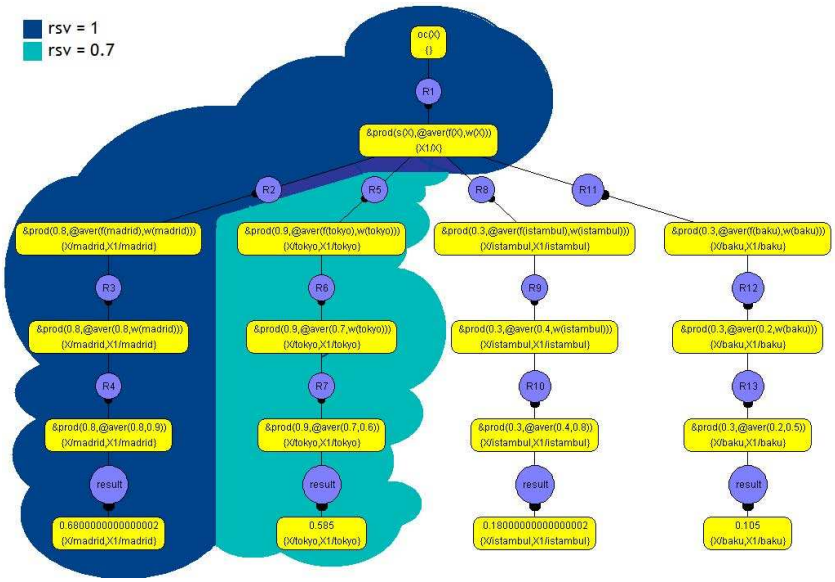
"`//node[children[not(text())] and rule/text()<>"result"]/goal`", helping us to know whether the tree has any partially evaluated leaf (i.e., non reporting a fuzzy computed answer) since it returns nodes at the end of a branch that are not labeled with the *rule* tag containing "`result`". The important meaning of this query resides on its capability for finding possible sources of infinite loops. For instance, if we work with a program containing a rule like "`p <- p`", when using FLOPER for generating an execution tree for "`p`" with any depth level, it will always contain at least a leaf reported by the previous *fuzzy XPath* query.

In order to take advantage of the enrichments introduced in the *fuzzy XPath* language, the following query makes use of "`DEEP`" and "`FILTER`" commands in order to perform a *partial breadth-first traversal* on execution trees as shown in Figure 8. In the resulting XML output, 10 nodes have been selected from the execution tree with different "`rsv`" values, varying from 1 in the case of the original goal (that has not been penalized) till 0.531441 for the fourth row, representing nodes whose depth ("DEEP-level") remains above the filter. Note that the use of

the directive "DEEP" segregates the nodes of the tree from top to bottom, since lower nodes in the tree are represented deeper in the input XML file.

Analogously, in Figure 9 we use "DOWN" instead of "DEEP" for producing *partial depht-first traversals* on execution trees. In this case, our query segregates the nodes from left to right in columns, since the more left the node appears in the tree, the upper is it in the XML output and, thus, the less penalized by "DOWN". As previously, 10 nodes have been selected again with "rsv" ranging from 1 -upper nodes in the XM file- in the left column, till 0.7, as shown in the second column.

Figure 9: Executing query «[FILTER=0.5][DOWN=0.7]//node/goal»

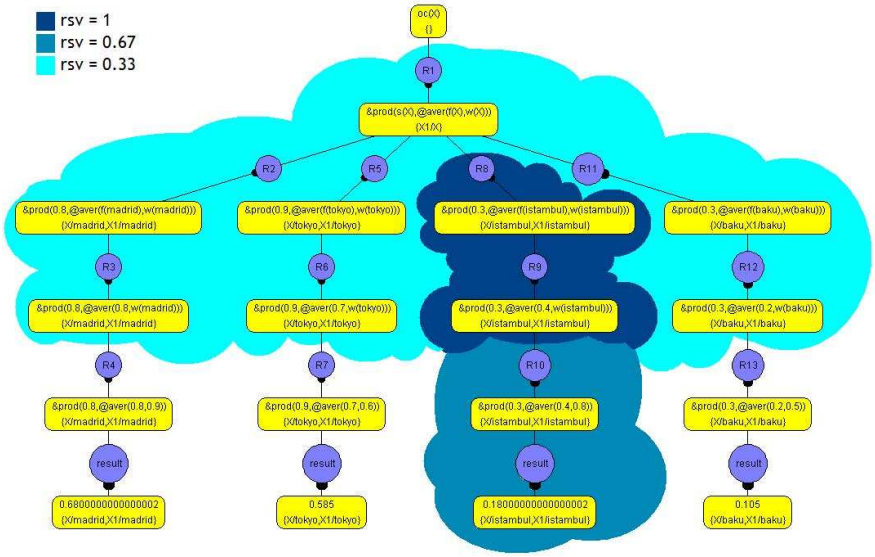



```
<result>
  <goal rsv="1.0">oc(X)</goal>
  <goal rsv="1.0">and_prod(s(X),agr_aver(f(X),w(X)))</goal>
  <goal rsv="1.0">and_prod(0.8,agr_aver(f(madrid),w(madrid)))</goal>
  <goal rsv="1.0">and_prod(0.8,agr_aver(0.8,w(madrid)))</goal>
  <goal rsv="1.0">and_prod(0.8,agr_aver(0.8,0.9))</goal>
  <goal rsv="1.0">0.6800000000000002</goal>
  <goal rsv="0.7">and_prod(0.9,agr_aver(f(tokyo),w(tokyo)))</goal>
  <goal rsv="0.7">and_prod(0.9,agr_aver(0.7,w(tokyo)))</goal>
  <goal rsv="0.7">and_prod(0.9,agr_aver(0.7,0.6))</goal>
  <goal rsv="0.7">0.585</goal>
</result>
```

In order to illustrate the high expressive power of the *fuzzy XPath* language, in the following we try to model queries joining several concepts (for instance, the topics of "weather" and "Istambul" modeled in $\mathscr{P}$ as predicate "*w*" and constant "*istambul*", respectively). Assume that we are firstly interested on nodes informing about "weather", i.e., focusing on the fourth rows of our execution tree, thus meaning that sub-string "w(" must appear in tag "*goal*", while our second

preference asks for nodes in the branch containing the word "`istambul`" in tag "*substitution*". In order to join these two constraints, instead of using crisp "*or/and*" operators (or even different fuzzy variants of such connectives already implemented in *fuzzy XPath*), we prefer to use an arithmetical average giving twice importance to the second requirement than to the first one. The *fuzzy XPath* formulation of our query entitles Figure 10, where we graphically show the set of solutions as well as the output in the resulting XML file.

Figure 10: «node[/goal[contains(text(),"w(")]aver{1,2}substitution[contains(text(),"istambul")]]//goal»




```
<result>
  <goal rsv="1.0">and_prod(0.3,agr_aver(f(istambul),w(istambul)))</goal>
  <goal rsv="1.0">and_prod(0.3,agr_aver(0.4,w(istambul)))</goal>
  <goal rsv="0.6667">and_prod(0.3,agr_aver(0.4,0.8))</goal>
  <goal rsv="0.6667">0.18000000000000002</goal>
  <goal rsv="0.3333">and_prod(s(X),agr_aver(f(X),w(X)))</goal>
  <goal rsv="0.3333">and_prod(0.8,agr_aver(f(madrid),w(madrid)))</goal>
  <goal rsv="0.3333">and_prod(0.9,agr_aver(f(tokyo),w(tokyo)))</goal>
  <goal rsv="0.3333">and_prod(0.3,agr_aver(f(baku),w(baku)))</goal>
  <goal rsv="0.3333">and_prod(0.8,agr_aver(0.8,w(madrid)))</goal>
  <goal rsv="0.3333">and_prod(0.9,agr_aver(0.7,w(tokyo)))</goal>
  <goal rsv="0.3333">and_prod(0.3,agr_aver(0.2,w(baku)))</goal>
</result>
```

5 Conclusions and Future Work

In this paper we have shown the mutual benefits between two different fuzzy tools developed in our research group, that is, the FLOPER programming environment and the *fuzzy XPath* interpreter. Initially FLOPER was conceived as a tool for implementing flexible software ap-

plications -as it is the case of *fuzzy XPath*- coded with the fuzzy logic language MALP and offering options for compiling fuzzy rules to standard Prolog clauses, running goals and drawing execution trees. Such trees, once modeled in XML format inside the proper FLOPER tool, can be then analyzed by the *fuzzy XPath* interpreter -by means of simple XPath queries augmented with fuzzy commands- in order to discover details (such as fuzzy computed answers, possible infinite branches and so on) of the computational behaviour of MALP programs after being executed into FLOPER. In this sense, we plan to integrate an option inside the FLOPER menu for allowing the possibility of performing debugging tasks based on *fuzzy XPath*.

On the other hand, in [ALM12b, ALM13] we have recently presented a *fuzzy XPath* debugger (beyond the *fuzzy XPath* interpreter) that, for a given XPath expression, the tool offers a set of alternative queries, each one associated to a chance degree indicating the deviations of each proposal w.r.t. the original query (we use `JUMP`, `DELETE` and `SWAP` operators for covering the main cases of programming errors when describing a path about an XML document). Thus, our tool is focused on providing the programmer a repertoire of paths that (s)he can use to retrieve answers. Since in this paper we have seen that the *fuzzy XPath* interpreter might act as a debugger of fuzzy computations developed with FLOPER, for the near future we plan too to study the role that the proper *fuzzy XPath* debugger should play for helping the development of applications using FLOPER.

Acknowledgements: This work has been partially supported by the EU, under FEDER, and the Spanish Science and Innovation Ministry (MICINN) under grant TIN2008-06622-C03-03, as well as by Ingenieros Alborada IDI under grant TRA2009-0309, and the JUNTA ANDALUCIA administration under grant TIC-6114 (proyecto de excelencia). Carlos Vázquez and Ginés Moreno received grants for International mobility from the University of Castilla-La Mancha (CYTEMA project and “Vicerrectorado de Profesorado”).

Bibliography

[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, 2011.

[ALM12a] 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.

[ALM12b] J. M. Almendros-Jiménez, A. Luna, G. Moreno. A XPath Debugger Based on Fuzzy Chance Degrees. In *On the Move to Meaningful Internet Systems: Proceedings OTM 2012 Workshops, Rome, Italy, September 10-14*. Pp. 669–672. Springer Verlag, LNCS 7567, 2012.

[ALM13] J. Almendros-Jiménez, A. Luna, G. Moreno. Annotating Fuzzy Chance Degrees when Debugging XPath Queries. In *Advances in Computational Intelligence - Proc*

of the 12th International Work-Conference on Artificial Neural Networks, IWANN 2013 (Special Session on Fuzzy Logic and Soft Computing Application), Tenerife, Spain, June 12-14. Pp. 300–311. Springer Verlag, LNCS 7903, Part II, 2013.

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

[BMP95] J. F. Baldwin, T. P. Martin, B. W. Pilsworth. *Fril- Fuzzy and Evidential Reasoning in Artificial Intelligence*. John Wiley & Sons, Inc., 1995.

[GMV04] S. Guadarrama, S. Muñoz, C. Vaucheret. Fuzzy Prolog: A New Approach Using Soft Constraints Propagation. *Fuzzy Sets and Systems* 144(1):127–150, 2004.

[KMP00] E. Klement, R. Mesiar, E. Pap. *Triangular Norms*. Trends in logic, Studia logica library. Springer, 2000.
http://books.google.es/books?id=rIyqcjfKMN4C

[KS92] M. Kifer, V. Subrahmanian. Theory of generalized annotated logic programming and its applications. *Journal of Logic Programming* 12:335–367, 1992.

[Llo87] J. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987.

[MCS11] S. Muñoz-Hernández, V. P. Ceruelo, H. Strass. RFuzzy: Syntax, semantics and implementation details of a simple and expressive fuzzy tool over Prolog. *Inf. Sci.* 181(10):1951–1970, 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.

[SS83] B. Schweizer, A. Sklar. *Probabilistic Metric Spaces*. Courier Dover Publ., 1983.
http://books.google.es/books?id=8LUd6Txuu5sC

[Voj01] P. Vojtáš. Fuzzy Logic Programming. *Fuzzy Sets and Systems* 124(1):361–370, 2001.