Fuzzy Logic Programming for Implementing a Flexible XPath-based Query Language

Jesús M. Almendros-Jiménez
Department of Languages and Computation, University of Almería, Spain

Alejandro Luna and Ginés Moreno
Department of Computing Systems, University of Castilla-La Mancha, Spain

Abstract

FLOPER is the “Fuzzy LOGic Programming Environment for Research” designed in our research group for assisting the development of real-world applications where fuzzy logic might play an important role. This is the case of our recently proposed extension for the popular XPath query language in order to handle flexible queries which provide ranked answers, fuzzy variants of operators and, or and avg for XPath conditions, as well as two structural constraints, called down and deep, for which a certain degree of relevance can be associated.

Keywords: Fuzzy Logic Programming, XPath Query Language, Software Tools

1 Introduction

The XPath language [7] has been proposed as a standard for XML querying and it is based on the description of the path in the XML tree to be retrieved. XPath allows to specify the name of nodes (i.e., tags) and attributes to be present in the XML tree together with boolean conditions about the content of

1 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, TIN 2007-65749 and TIN2011-25846, as well as by the Castilla-La Mancha Administration under grant PII1I09-0117-4481.

2 Email: jalmen@ual.es

3 Email: {gines.moreno,alejandro.luna}@uclm.es

This paper is electronically published in Electronic Notes in Theoretical Computer Science URL: www.elsevier.nl/locate/entcs
nodes and attributes. XPath querying mechanism is based on a boolean logic: the nodes retrieved from an XPath expression are those matching the path of the XML tree. Therefore, the user should know the XML schema in order to specify queries. However, even when the XML schema exists, it can not be available for users. Moreover, XML documents with the same XML schema can be very different in structure. Let us suppose the case of XML documents containing the curriculum vitae of a certain group of persons. Although they can share the same schema, each one can decide to include studies, jobs, training, etc. organized in several ways: by year, by relevance, and with different nesting degree.

Therefore, in the context of semi-structured databases, the need for flexible query languages arises, in which the user can formulate queries without taking into account a rigid schema database. In addition, they should be equipped with a mechanism for obtaining a certain ranked list of answers. The ranking of answers can provide satisfaction degree depending on several factors. In a structural XPath-based query, the main criteria to provide a certain degree of satisfaction depends on the hierarchical deepness and document order. Therefore the query language should provide mechanisms for assigning priority to answers when they occur in different parts of the document.

In this paper we focus on implementation issues based on fuzzy logic programming regarding our extension of the XPath query language initially presented in [5] for the handling of flexible queries. Our approach proposes two structural constraints called down and deep for which a certain degree of relevance can be 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 set of answers depending on the path they are found from “left to right” in the XML document. Both structural constraints can be combined. In addition, we provide fuzzy operators and, or and avg for XPath conditions. In this way, users can express the priority they give to answers. Such fuzzy operators can be combined to provide ranked answers. Our approach has been implemented by means of multi-adjoint logic programming and the FLOPER tool [16,17,18].

The need for providing flexibility to XPath has recently motivated the investigation of extensions of the XPath language. The most relevant ones are [8,9] in which authors introduce in XPath flexible matching by means of fuzzy constraints called close and similar for node content, together with below and near for path structure. In addition, they have studied deep-similar notion for tree matching. In order to provide ranked answers they adopt a Fuzzy set theory-based approach in which each answer has an associated numeric value (the membership degree). The numeric value represents the Retrieval Status Value (RSV) of the associated item. In the work of [11], they propose a satisfaction degree for XPath expressions based on associating a
Almendros, Luna and Moreno

degree of importance to XPath nodes, and they study how to compute the best $k$ answers. In both cases, authors allow the user to specify in the query the degree to which the answers will be penalized. On the other hand, in [10], they have studied how to relax XPath queries by means of rewriting in order to improve information retrieval in the presence of heterogeneous data resources.

Our work is similar to the proposed by [8,9]. The below operator of [8,9] is equivalent to our proposed down: both extract elements that are direct descendants of the current node, and the penalization is proportional to the distance. The near operator of [8,9], which is defined as a generalization of below, ranks answers depending of the distance to the required node, in any XPath axis. Our proposed deep ranks answers depending of the distance to the current node, but the nodes considered can be direct and non direct descendants. Therefore our proposed deep combined with down is a particular case of near. However, our aim is to extend the number of constraints and fuzzy operators of our approach thanks to the expressivity power of our framework based on fuzzy logic programming. The so-called multi-adjoint logic programming approach, MALP in brief [15], is an extension of logic programming to support fuzzy logic. Such framework provides theoretical basis for defining flexibility to XPath in many directions. In addition, the framework provides a mechanism for customizing ranked answers by assigning priorities to solutions independently of their occurrences.

With respect to similar and close operators proposed in [8,9], our framework lacks similarity relations and rather focuses on structural (i.e. path-based) flexibility. With regard to tree matching, the operator deep-similar defined in [8,9] can be simulated by means of deep and down operators. We believe that we could also work in the future in adapting our framework for working with degree of importance to XPath nodes along the lines of [11], and relaxing XPath expressions by rewriting in the line [10]. In both cases, our framework could provide ranked answers w.r.t. the degree of importance, and degree of matching. Our proposal makes use of the multi-adjoint logic programming framework for defining new fuzzy operators for XPath: and, or and avg. Such operators are used in XPath conditions on nodes and attribute values. They provide fuzzy combinations for ranking answers.

Finally, let us remark that our work is an extension of previous works about the implementation of XPath by means of logic programming [4], which has been extended to XQuery in [1]. The proposed extension follows the same encoding proposed in [1] in which a predicate called xpath is defined by means of Prolog rules, which basically traverse the Prolog representation of the XML tree by means of a Prolog list. In order to implement the flexible extension of XPath by means of the «Fuzzy Logic Programming Environment for Research» FLOPER (which is devoted to the management of MALP programs
we proceed similarly to the Prolog implementation of XPath, but proposing a new (fuzzy) predicate called fuzzyXPath implemented in MALP. The new query language returns a set of ranked answers each one with an associated RSV. Such RSV is computed by easily using MALP rules (thus exploiting the correspondences between the languages for-being and to-be implemented), where the notion of RSV is modeled inside a multi-adjoint lattice, and usual fuzzy connectives of the MALP language act as ideal resources to represent new flexible XPath operators. For instance, as we will detail in Section 4, the implementation of the main predicate fuzzyXPath in our application uses as parameters the user-proposed values for deep and down to filter/rank the set of desired answers, whereas new operators to model flexible XPath conditions admit a natural, elegant and direct representation via standard fuzzy connectives of MALP.

The structure of the paper is as follows. Whereas in Section 2 we present our fuzzy extension of XPath, Section 3 is devoted to the formal/practical description of the multi-adjoint logic programming framework as well as the FLOPER environment. Next, Section 4 discusses the implementation details of our approach and finally, Section 5 concludes planning future work.

2 Flexible XPath

Our flexible XPath is defined by means of the following rules:

\[
\begin{align*}
\text{xpath} & : = \text{[deepdown]} \text{path} \\
\text{path} & : = \text{literal} \mid \text{text()} \mid \text{node} \mid \text{@att} \mid \\
& \quad \text{node/path} \mid \text{node//path} \\
\text{node} & : = \text{QName} \mid \text{QName[cond]} \\
\text{cond} & : = \text{path op path} \\
\text{deepdown} & : = \text{DEEP=degree,DOWN=degree} \\
\text{op} & : = \text{>} \mid \text{=} \mid \text{<} \mid \text{and} \mid \text{or} \mid \text{avg}
\end{align*}
\]

Basically, our proposal extends XPath as follows:

- A given XPath expression can be adorned with «[DEEP = \text{r}_1, \text{DOWN} = \text{r}_2]» which means that the deepness of elements is penalized by \text{r}_1 and that the order of elements is penalized by \text{r}_2, and such penalization is proportional to the distance. In particular, «[DEEP = 1, DOWN = \text{r}_2]» can be used for penalizing only w.r.t. document order. DEEP works for //, and DOWN works for / and //.
- Moreover, the classical and and or connectives admit here a fuzzy behavior
based on fuzzy logic, i.e., assuming two given RSVs $r_1$ and $r_2$, operator $\text{and}$ is defined as $r_3 = r_1 \ast r_2$ and operator $\text{or}$ returns $r_3 = r_1 + r_2 - (r_1 \ast r_2)$. In addition, the $\text{avg}$ operator is defined as $r_3 = (r_1 + r_2)/2$.

In general, an extended 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 1, whose skeleton is depicted in Figure 2.

Example 2.1 Consider the XPath query: «[DEEP=0.9,DOWN=0.8]//title», that requests title’s penalizing the occurrences from the document root
Fig. 2. XML skeleton represented as a tree

Fig. 3. Output of a query using DEEP/DOWN

<table>
<thead>
<tr>
<th>Document</th>
<th>RSV computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;result&gt;</td>
<td>0.81 = 0.9²</td>
</tr>
<tr>
<td>&lt;title rsv=&quot;0.81&quot;&gt;Don Quijote de la Mancha&lt;/title&gt;</td>
<td>0.52488 = 0.9⁴ × 0.8</td>
</tr>
<tr>
<td>&lt;title rsv=&quot;0.52488&quot;&gt;La Galatea&lt;/title&gt;</td>
<td>0.340122 = 0.9⁶ × 0.8²</td>
</tr>
<tr>
<td>&lt;title rsv=&quot;0.340122&quot;&gt;Los trabajos de Persiles y Sigismunda&lt;/title&gt;</td>
<td>0.648 = 0.9² × 0.8</td>
</tr>
<tr>
<td>&lt;title rsv=&quot;0.648&quot;&gt;La Celestina&lt;/title&gt;</td>
<td>0.5184 = 0.9² × 0.8²</td>
</tr>
<tr>
<td>&lt;title rsv=&quot;0.5184&quot;&gt;Hamlet&lt;/title&gt;</td>
<td>0.335923 = 0.9⁴ × 0.8³</td>
</tr>
<tr>
<td>&lt;title rsv=&quot;0.335923&quot;&gt;Romeo y Julieta&lt;/title&gt;</td>
<td>0.41472 = 0.9² × 0.8³</td>
</tr>
<tr>
<td>&lt;title rsv=&quot;0.41472&quot;&gt;Las ferias de Madrid&lt;/title&gt;</td>
<td>0.268739 = 0.9⁴ × 0.8⁴</td>
</tr>
<tr>
<td>&lt;title rsv=&quot;0.268739&quot;&gt;El remedio en la desdicha&lt;/title&gt;</td>
<td>0.214991 = 0.9⁴ × 0.8⁵</td>
</tr>
<tr>
<td>&lt;title rsv=&quot;0.214991&quot;&gt;La Dragontea&lt;/title&gt;</td>
<td>0.214991 = 0.9⁴ × 0.8⁵</td>
</tr>
<tr>
<td>&lt;result&gt;</td>
<td>0.5 = (0 + 1)/2</td>
</tr>
<tr>
<td>&lt;book rsv=&quot;0.5&quot;&gt;...&lt;/book&gt;&lt;title&gt;Don Quijote ...&lt;/title&gt;</td>
<td>1 = (1 + 1)/2</td>
</tr>
<tr>
<td>&lt;book rsv=&quot;1.0&quot;&gt;...&lt;/book&gt;&lt;title&gt;La Celestina&lt;/title&gt;</td>
<td>1 = (1 + 1)/2</td>
</tr>
<tr>
<td>&lt;book rsv=&quot;1.0&quot;&gt;...&lt;/book&gt;&lt;title&gt;Hamlet&lt;/title&gt;</td>
<td>1 = (1 + 1)/2</td>
</tr>
<tr>
<td>&lt;book rsv=&quot;0.5&quot;&gt;...&lt;/book&gt;&lt;title&gt;Las ferias de Madrid&lt;/title&gt;</td>
<td>0.5 = (1 + 0)/2</td>
</tr>
<tr>
<td>&lt;result&gt;</td>
<td>0.5 = (0 + 1)/2</td>
</tr>
</tbody>
</table>

Fig. 4. Output of a query using the average operator AVG
by a proportion of 0.9 and 0.8 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 publication’s).

Example 2.2 Figure 4 shows the answer associated to the XPath expression: «/bib/book[@price<30 avg @year<2006]». Here we show that books satisfying a price under 30 and a year before 2006 have the highest RSV.

Example 2.3 Finally, in Figure 5 we combine all operators (thus obtaining more scattered RSV values) in query: «[DEEP=0.9,DOWN=0.8] //book [(@price>25 and @price<30) avg (@year<2000 or @year>2006)]/title».

3 Multi-Adjoint Logic Programming and FLOPER

Logic Programming (LP) [14] has been widely used as a formal method for problem solving and knowledge representation in the past. Nevertheless, traditional LP languages do not incorporate techniques or constructs to deal explicitly with uncertainty and approximated reasoning. To overcome this situation, during the last decades several fuzzy logic programming systems have been developed where the classical inference mechanism of SLD–Resolution have been replaced with a fuzzy variant able to handle partial truth and to reason with uncertainty. Most of these systems implement the fuzzy resolution principle introduced by Lee in [13], such as languages Prolog-Elf [12], Fril [6], the QLP scheme of [19], the many-valued logic programming language of [21,20] and MALP [15]. In this paper we are mainly concerned with this last framework, which uses a syntax near to Prolog but enjoys higher levels of flexibility and for which we are developing the FLOPER tool (see [16,17,18] and visit http://dectau.uclm.es/floper). In what follows, we present a short summary of the main features of MALP (we refer the reader to [15] for a complete formulation).
We work with a first order language, $\mathcal{L}$, containing variables, function symbols, predicate symbols, constants, quantifiers ($\forall$ and $\exists$), and several arbitrary connectives such as implications ($\leftarrow_1, \leftarrow_2, \ldots, \leftarrow_m$), conjunctions ($\&_1, \&_2, \ldots, \&_k$), disjunctions ($\lor_1, \lor_2, \ldots, \lor_l$), 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 $\mathcal{L}$ contains the values of a multi-adjoint lattice, $\langle L, \preceq, \leftarrow_1,\ldots,\leftarrow_n,\&_1,\ldots,\&_n \rangle$, 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 “$A \leftarrow B$ with $\alpha$”, where $A$ is an atomic formula (usually called the *head*), $B$ (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 $\leq$. Consider, for instance, the following program $\mathcal{P}$ composed of three rules with associated multi-adjoint lattice $\langle [0, 1], \leq, \leftarrow_p, \&_p \rangle$, where label $\mathcal{P}$ mean for *Product logic* with the following connective definitions (for implication, conjunction and disjunction symbols, respectively): “$\leftarrow_p (x, y) = \min(1, x/y)$”, “$\&_p(x, y) = x \ast y$” and “$\mathcal{P}(x, y) = x + y - x \ast y$”.

$\begin{align*}
\mathcal{R}_1 : \quad & p(X) \leftarrow_p q(X, Y) \mid_p r(Y) \quad \text{with} \quad 0.8 \\
\mathcal{R}_2 : \quad & q(a, Y) \leftarrow \quad \text{with} \quad 0.9 \\
\mathcal{R}_3 : \quad & r(b) \leftarrow \quad \text{with} \quad 0.7
\end{align*}$

In order to describe the procedural semantics of the multi-adjoint logic language, in the following we denote by $\mathcal{C}[A]$ a formula where $A$ is a sub-expression (usually an atom) which occurs in the (possibly empty) one hole context $\mathcal{C}[\cdot]$ whereas $\mathcal{C}[A/A']$ means the replacement of $A$ by $A'$ in context $\mathcal{C}[\cdot]$, and $\text{mgu}(E)$ is the most general unifier of an equation set $E$. The pair $(\mathcal{Q}; \sigma)$ composed of a goal and a substitution is called a *state*. So, given a program $\mathcal{P}$, an admissible computation is formalized as a state transition system, whose transition relation $\overset{\mathcal{AS}}{\Rightarrow}$ is the smallest relation satisfying the following admissible rules:

1) $(\mathcal{Q}[A]; \sigma) \overset{\mathcal{AS}}{\Rightarrow} (\mathcal{Q}[A/v\&\mathcal{B}]\theta; \sigma\theta)$ if $A$ is the selected atom in goal $\mathcal{Q}$, $(A'\leftarrow B \text{ with } v)$ in $\mathcal{P}$, where $\mathcal{B}$ is not empty, and $\theta = \text{mgu}(\{A' = A\})$.

2) $(\mathcal{Q}[A]; \sigma) \overset{\mathcal{AS}}{\Rightarrow} (\mathcal{Q}[A/v]\theta; \sigma\theta)$ if $(A'\leftarrow \text{ with } v)$ in $\mathcal{P}$, $\theta = \text{mgu}(\{A' = A\})$.

The following derivation illustrates our definition (note that the exact program rule used -after being renamed- in the corresponding step is annotated as a super–index symbol, whereas exploited atoms appear underlined):
The final formula can be directly interpreted in the lattice \( L \) to obtain the final fuzzy computed answer. So, since \( 0.8 \land_p (0.9 \mid_p 0.7) = 0.8 \ast (0.9 + 0.7) - (0.9 \ast 0.7) = 0.776 \), we say that goal \( p(X) \) is true at a 77.6% when \( X = a \).

The parser of our FLOPER tool [16,17,18] 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 is to extend each atom with an extra argument, called \textit{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 clause in our target program is translated into:

\[
\begin{align*}
\text{p}(X, & \_TV0) :- q\big(X,Y,\_TV1\big), r\big(Y,\_TV2\big), \text{or\_prod}(\_TV1,\_TV2,\_TV3), \text{and\_prod}(0.8,\_TV3,\_TV0).
\end{align*}
\]

Moreover, the remaining rules in our fuzzy program, become the pure Prolog facts “q(a,Y,0.9)” and “r(b,0.7)”, whereas the corresponding lattice is expressed by these clauses (the meaning of the mandatory predicates \texttt{member}, \texttt{top}, \texttt{bot} and \texttt{leq} is obvious):

\[
\begin{align*}
\text{member}(X) :- & \text{number}(X), 0=<X,X=<1. \\
\text{bot}(0). \\
\text{top}(1). \\
\text{leq}(X,Y) :- & X=<Y. \\
\text{or\_prod}(X,Y,Z) :- & \text{pri\_prod}(X,Y,U1),\text{pri\_add}(X,Y,U2),\text{pri\_sub}(U2,U1,Z). \\
\text{and\_prod}(X,Y,Z) :- & \text{pri\_prod}(X,Y,Z). \\
\text{pri\_prod}(X,Y,Z) :- & Z = X \ast Y. \\
\text{pri\_add}(X,Y,Z) :- & Z = X+Y. \\
\text{pri\_sub}(X,Y,Z) :- & Z = X-Y.
\end{align*}
\]

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 option “run”, the Prolog interpreter returns the desired fuzzy computed answer \( \text{Truth\_degree} = 0.776, X = a \). 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 (visualize) 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 \texttt{member}, \texttt{top}, \texttt{bot}, \texttt{leq} and the ones associated to fuzzy connectives) in order to be loaded into FLOPER.
3.1 MALP and XPath

MALP can be used as basis for our proposed flexible extension of XPath as follows. The idea is to implement XPath by means of MALP rules. With this aim, firstly, we have to consider a complete lattice $L_{\text{tv}}$ to be used for representing RSVs associated to elements of an XML tree. $L_{\text{tv}}$ contains trees, which we will call $\text{tv}$ trees, of the form: "$\text{tv}(rsv, [\text{root}, \text{tvch}, \text{tvsib}])$", where $rsv$ is a value taken from $[0,1]$, $\text{root}$ is the root of the tree to which $rsv$ is associated, and $\text{tvch}$, $\text{tvsib}$ are the $\text{tv}$ trees of the children and sibling nodes. Such $\text{tv}$ trees are the answer of the MALP program to a XPath expression. In this way, MALP is able to compute the RSVs associated to each node of the input XML tree.

For such a lattice $L_{\text{tv}}$ we can define the partial ordering $\preceq$ induced from the the partial order of $[0,1]$: $\text{tv}(r_1, \text{tree}_1) \preceq \text{tv}(r_2, \text{tree}_2)$ iff $r_1 \preceq r_2$. Using such lattice, the MALP rules have the form "$A \leftarrow B$ with $\text{tv}$tree". In addition, the XPath fuzzy operators $\text{and}$ and $\text{or}$ and $\text{avg}$ can be mapped to MALP connectives in lattice $L_{\text{tv}}$. Finally, we have to consider an auxiliary aggregator in $L_{\text{tv}}$ called $\text{@fuse}$, which builds the $\text{tv}$ tree of a certain node from the $\text{tv}$ trees of the sibling and children nodes. In Figure 6, we can see the lattice defined by means of Prolog syntax (to be loaded into FLOPER), where $\text{and_pro}$, $\text{or_prod}$ and $\text{agr_aver}$ represent the XPath operators $\text{and}$, $\text{or}$ and $\text{avg}$, respectively. Let us remark that they are defined for $\text{tv}$ trees and use in their definitions the operators $\text{and}$, $\text{or}$ and $\text{avg}$ of the lattice $[0,1]$. 

Figure 6. Multi-adjoint lattice for “fuzzyXPath” (file “tv.pl”)
4 Implementation

Although the core of our application is written with (fuzzy) MALP rules, we have reused/adapted several modules of our previous Prolog-based implementation of (crisp) XPath described in [1,2,3,4], which make use of the SWI-Prolog library for loading XML files in order to store each XML document by means of a Prolog term\(^5\) representing a tree. The clever idea is that each tag is represented as a data-term of the form

\[
\text{element}(\text{Tag}, \text{Attributes}, \text{Subelements})
\]

where \text{Tag} is the name of the XML tag, \text{Attributes} is a list containing the attributes, and \text{Subelements} is a list containing the subelements (i.e. subtrees) of the tag. For instance, let us consider the XML document of Figure 1, represented in SWI-Prolog like in Figure 7. Moreover, for loading XML documents in our prototype we can use the predicate \text{load_xml}(\text{+File,-Term}) defined as follows:

\[
\text{load_xml}(\text{File}, \text{Term}):=\text{load_structure}(\text{File}, \text{Term}, [\text{dialect}(\text{sgml})])
\]

where \text{load_structure}(\text{+File,-Term,+Options}) is the SWI-Prolog predicate of the XML library for loading XML documents. Similarly, we have implemented a predicate called \text{write_xml}(\text{+File,+Term}) for writing data-terms representing an XML document into a file. And, of course, the parser of our application has been extended to recognize the new keywords \text{deep, down, avg, etc...} with their proper arguments.

Now, we are going to present how the new «fuzzyXPath» predicate admits an elegant definition by means of fuzzy MALP rules which, after being compiled into clauses using FLOPER, can be safely executed in any standard Prolog platform. Each rule defining predicate

\[
\text{fuzzyXPath}(\text{ListXPath}, \text{Tree}, \text{Deep}, \text{Down})
\]

\(^5\) The notion of \text{term} (i.e., data structure) is just the same in MALP and Prolog.
receives four arguments: (1) \texttt{ListXPath} is the Prolog representation of an XPath expression, (2) \texttt{Tree} is the term representing an input XML document and (3) \texttt{Deep/Down} which have the obvious meaning -their default values are \texttt{tv(1,[]} numeral-6. A call to this predicate, returns after being executed a truth-value (i.e., a \texttt{tv tree}) of the one depicted in Figure 8.

For instance, the query «[\texttt{Deep}=0.9,\texttt{Down}=0.8]/Path » on a given XML term, would generate a call of the form

\[
\texttt{fuzzyXPath(Path, XML, tv(0.9, []), tv(0.8, []))}
\]

whose further execution will return the resulting \texttt{tv tree}.

Basically, the fuzzyXPath predicate traverses the Prolog tree representing an XML document and extracts in the returned \texttt{tv tree} the subtrees occurring

\[6\] These parameters could be avoided if we declare \texttt{deep} and \texttt{down} as \textit{constants} in the lattice (in a similar way as done, for instance, with \texttt{bot} and \texttt{top} in Figure 6) but in that case we would need to redefine them at the beginning of each query evaluation. So, we prefer our present option which is easy to understand and safe, in the sense that in the Prolog code generated by \textsc{FLOPER} when compiling MALP programs, the notions of truth-degree and fuzzy connectives are assimilated to data-terms and predicates, respectively.
in the given path, also annotating into the nested tv trees the corresponding deep/down values according to the movements performed (in the horizontal and vertical axis, respectively) when navigating on the XML tree.

The definition of such predicate includes several rules for distinguishing cases in the form of the input document and the XPath expression. As an example, we can see the rule of Figure 9, whose translation to Prolog is shown in Figure 10.

Let us remark that the parameters Deep and Down are tv trees of the form \( \text{tv}(r_1,[]) \) and \( \text{tv}(r_2,[]) \) whenever the XPath expression is of the form \( [DEEP = r_1, DOWN = r_2]/\cdots \). They are passed as arguments in order to be used for penalizing nesting and order.

Let us explain in detail the fuzzy code of Figure 9. After performing a recursive call to compute the solutions associated to the Children of a given node (i.e., \( \text{fuzzyXPath}([\text{Label}|\text{LabelRest}], [\text{element}(\text{Label},_,\text{Children})|\text{Siblings}], \text{Deep}, \text{Down}) < \prod \)), we use connective &prod to muffle the resulting tv tree according to Deep, which is represented by sentence &prod(Deep, fuzzyXPath([Label|LabelRest], Children, Deep, Down)). A similar operation is next performed on the siblings of the node, whose result is penalized now according Down, that is, &prod(Down, fuzzyXPath([Label|LabelRest], Siblings, Deep, Down)). Finally, both tv trees are combined (fused) with the content of the current node.

On the other hand, when considering queries containing expressions with conditions, the fuzzyXPath predicate evolves as shown in the MALP rule listed in Figure 11. Here it is remarkable the direct use of connective avg when defining the recursive execute_fcond predicate.

Finally, we have defined a predicate tv_to_elem to show the result in a pretty way (see Figures 3 and 4), which transforms the returned tv tree to an XML tree.
Fig. 11. Schema of MALP rules for evaluating conditions (with avg)

```
fuzzyXPath([Label,tree(A,B,C)],[element(Label,Attr,Children)|Siblings], Deep, Down) <prod
  @fuse(
    execute_fcond(Label, tree(A,B,C), element(Label,Attr,Children)),
    &prod(Down, fuzzyXPath([Label, tree(A,B,C)], Siblings, Deep, Down))
  ) with tv(1,[])
```

```
execute_fcond(Label, tree(avg, T1, T2), element(Label,Attr,Children)) <prod
  @avg(
    execute_fcond(Label, T1, element(Label,Attr,Children)),
    execute_fcond(Label, T2, element(Label,Attr,Children))
  ) with tv(1,[])
```

5 Conclusions and Future Work

In this paper we have completed the preliminary description of our flexible extension of XPath we initially presented in [5], but focusing now on implementation issues which were omitted in this last work (our prototype can be found in: http://dectau.uclm.es/fuzzyXPath/). The material presented here represents the first real-world application developed with the fuzzy logic language MALP (using too our FLOPER tool), by showing its capabilities for easily modeling scenarios where concepts somehow based on fuzzy logic play a crucial role. In particular, we have shown the ability of the MALP language for easily coding the new constructs (both structural -deep and down- and constraints -avg and fuzzy versions of classical or/and operators-) of the enriched dialect of XPath, in order to flexibly query XML documents.

We are currently working on some extensions suggested by the power of MALP regarding two main implementation lines: a) defining commands for “reverse axes” such as up, i.e. the counterpart of down (sometimes the more nested information is the more basic and should receive a better relevance), thus connecting with the classical near operator and b) using a whole family of fuzzy connectives (belonging, for instance, to the well known Gödel and Łukasiewicz fuzzy logics for describing scenarios with a somehow optimistic/pessimistic taste) in order to express more flexible conditions on fuzzyXPath queries. We think that this research line promises fruitful developments in the near future by reinforcing the power of fuzzy XPath commands, extensions to cope with XQuery and the semantic web, etc.

References


