Fuzzy XPath for the Automatic Search of Fuzzy Formulae Models*

Jesús M. Almendros-Jiménez[1], Miquel Bofill[2], Alejandro Luna-Tedesqui[3], Ginés Moreno[3], Carlos Vázquez[3], and Mateu Villaret[2]

[1] Dept. Languages and Computation, U. Almería (Spain)
[2] Dept. Computer Science, Applied Mathematics and Statistics, U. Girona (Spain)
[3] Dept. Computing Systems, U. Castilla-La Mancha, Albacete (Spain)
jalmen@ual.es,{Miquel.Bofill,Mateu.Villaret}@udg.edu,
{Alejandro.Luna,Gines.Moreno,Carlos.Vazquez}@uclm.es

Abstract. In this paper we deal with propositional fuzzy formulae containing several propositional symbols linked with connectives defined in a lattice of truth degrees more complex than *Bool*. Instead of focusing on satisfiability (i.e., proving the existence of at least one model) as usually done in a SAT/SMT setting, our interest moves to the problem of finding the whole set of models (with a finite domain) for a given fuzzy formula. We re-use a previous method based on fuzzy logic programming where the formula is conceived as a goal whose derivation tree, provided by our FLOPER tool, contains on its leaves all the models of the original formula, together with other interpretations. Next, we use the ability of the FuzzyXPath tool (developed in our research group with FLOPER) for exploring these derivation trees once exported in XML format, in order to discover whether the formula is a tautology, satisfiable, or a contradiction, thus reinforcing the bi-lateral synergies between FuzzyXPath and FLOPER.

Keywords: Fuzzy Logic Programming; Automatic Theorem Proving; Fuzzy XPath

1 Introduction

Research on SAT (Boolean Satisfiability) and SMT (Satisfiability Modulo Theories) [7] represents a successful and large tradition in the development of highly efficient automatic theorem solvers for classic logic. More recently there also exist attempts for covering fuzzy logics, as occurs with the approaches presented in [6, 15]. Moreover, if automatic theorem solving supposes too a starting point for the foundations of logic programming as well as one of its important application fields [11, 14], in [9] we showed some preliminary guidelines about how fuzzy

* This work has been partially supported by the EU (FEDER), and the Spanish MINECO Ministry (*Ministerio de Economía y Competitividad*) under grants TIN2013-44742-C4-4-R, TIN2012-33042 and TIN2013-45732-C4-2-P.

logic programming can face the automatic proving of fuzzy theorems by making use of the FLOPER environment developed in our research group [13] (visit `http://dectau.uclm.es/floper/`). The main goal of the present paper is to reinforce this last method of [9] by means of the FuzzyXPath tool developed too with FLOPER as described in [2, 5, 4] (the application is freely available from `http://dectau.uclm.es/fuzzyXPath`).

Let us start our discussion with an easy motivating example. Assume that we have a very simple digital chip with just a single input port and just one output port, such that it reverts on Out the signal received from In. The behaviour of such chip can be represented by the following propositional formula $F : (\neg In \wedge Out) \vee (In \wedge \neg Out)$. Depending on how we interpret each propositional symbol, we obtain the following final set of interpretations for the whole formula:

$$I1 : \{In = 0, Out = 0\} \Rightarrow F = 0 \qquad I2 : \{In = 0, Out = 1\} \Rightarrow F = 1$$
$$I3 : \{In = 1, Out = 0\} \Rightarrow F = 1 \qquad I4 : \{In = 1, Out = 1\} \Rightarrow F = 0$$

A SAT solver easily proves that F is satisfiable since, in fact, it has two models (i.e., interpretations of the propositional variables In and Out that assign 1 to the whole formula) represented by $I2$ and $I3$. An alternative way for explicitly obtaining such interpretations consists of using the fuzzy logic environment FLOPER developed in our research group. As we will explain in the rest of the paper, when FLOPER executes the following goal representing formula F "`(@not(i(In)) & i(Out)) | (i(In) & @not(i(Out)))`" with respect to a fuzzy logic program composed by just two rules: "`i(1) with 1`" and "`i(0) with 0`", it generates an execution tree where models $I2$ and $I3$ appear as leaves (see [9]). Each branch in the tree starts by interpreting variables In and Out and continues with the evaluation of operators (connectives) appearing in F.

Note that whereas formula F describes the behaviour of our chip in an "implicit way", the whole set of models $I2$ and $I3$ "explicitly" describes how the chip successfully works (any other interpretation not being a model, represents an abnormal behaviour of the chip), hence the importance of finding the whole set of models for a given formula.

Assume now that we plan to model an "analogic" version of the chip, where both the input and output signals might vary in an infinite range of values between 0 and 1, such that Out will simply represent the "complement" of In. The new behaviour of the chip can be expressed again by the same previous formula, but taking into account now that connectives involved in F could be defined in a fuzzy way as follows (see also Figure 1 afterwards):

$$\begin{array}{ll} \neg x = 1 - x & \text{Product logic's negation} \\ x \wedge y = min(x, y) & \text{Gödel logic's conjunction} \\ x \vee y = min(x + y, 1) & \text{Łukasiewicz logic's disjunction} \end{array}$$

Here we could use an SMT solver to prove that F is satisfiable, as done in [6, 9], but the goal of this paper is to use techniques based on fuzzy logic programming for discovering models.

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 [8] 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 [2, 5, 4] we have presented an XPath interpreter (together with a debugger, as documented in [1, 3]) extended with fuzzy commands which somehow rely on the implementation based on fuzzy logic programming by using FLOPER.

In [5] we illustrated the mutual benefits between the FLOPER programming environment and the FuzzyXPath interpreter. Initially FLOPER was conceived as a tool for implementing flexible software applications –as it is the case of FuzzyXPath– 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 FuzzyXPath 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. The main goal of this paper is to use FuzzyXPath for to automate the process of directly extracting the set of models contained on the proof trees associated to fuzzy formulae explained before, once such trees have been exported by FLOPER in XML format.

2 Fuzzy Logic Programming and FLOPER

In what follows we describe a very simple *subset of the Multi-Adjoint Logic Programming language*, MALP in brief, (see [12]for a complete formulation of this framework), which in essence consists of a first-order language, $\mathcal{L}$, containing variables, constants, function symbols, predicate symbols, and several (arbitrary) connectives to increase language expressiveness: implication connectives (denoted by $\leftarrow_1, \leftarrow_2, \ldots$); conjunctive connectives ($\wedge_1, \wedge_2, \ldots$), disjunctive connectives ($\vee_1, \vee_2, \ldots$), and hybrid operators (usually denoted by $@_1, @_2, \ldots$), all of them are grouped under the name of "aggregators". Although these connectives are binary operators, we usually generalize them as functions with an arbitrary number of arguments. So, we often write $@(x_1, \ldots, x_n)$ instead of $@(x_1, \ldots, @(x_{n-1}, x_n))$. By definition, the truth function for an n-ary aggregation operator $[\![@]\!] : L^n \to L$ is required to be monotonous.

$$
\begin{array}{lll}
\&_{\mathtt{P}}(x,y) \triangleq x * y & |_{\mathtt{P}}(x,y) \triangleq x + y - x * y & \leftarrow_{\mathtt{P}}(x,y) \triangleq \min(1, x/y) \\
\&_{\mathtt{G}}(x,y) \triangleq \min(x,y) & |_{\mathtt{G}}(x,y) \triangleq \max\{x,y\} & \leftarrow_{\mathtt{G}}(x,y) \triangleq \begin{cases} 1 & \text{if } y \leq x \\ x & \text{otherwise} \end{cases} \\
\&_{\mathtt{L}}(x,y) \triangleq \max(0, x+y-1) & |_{\mathtt{L}}(x,y) \triangleq \min\{x+y, 1\} & \leftarrow_{\mathtt{L}}(x,y) \triangleq \min\{x-y+1, 1\}
\end{array}
$$

Fig. 1: Conjunctors, disjunctors and implications from *Product*, *Gödel* and *Łukasiewicz* logics.

Additionally, our language $\mathcal{L}$ contains the values of a lattice $(L, \leq)$ and a set of connectives interpreted over such lattice. In general, L may be the carrier of any complete bounded lattice where a L-expression is a well-formed expression composed by values of L, as well as variable symbols, connectives and *primitive operators* (i.e., arithmetic symbols such as $*, +, min$, etc.). In what follows, we assume that the truth function of any connective @ in L is given by its corresponding *connective definition*, that is, an equation of the form $@(x_1, \ldots, x_n) \triangleq E$, where E is a L-expression not containing variable symbols apart from $x_1, \ldots, x_n$. For instance, some fuzzy connective definitions in the lattice $([0,1], \leq)$ are presented in Figure 1 (from now on, this lattice will be called $\mathcal{V}$ along this paper), where labels `L`, `G` and `P` mean respectively *Łukasiewicz logic*, *Gödel logic* and *product logic* (with different capabilities for modeling *pessimistic*, *optimistic* and *realistic scenarios*, respectively).

This subset of MALP is intended to cope with fuzzy propositional formulae like $P \wedge Q \rightarrow P \vee Q$, where propositions P and Q are interpreted as values of the lattice. To this end, a *program* is defined as a set of rules (also called "facts") of the form "H *with* v", where H is an atomic formula or atom (usually called *head*), and v is its associated *truth degree* (i.e., a value of L). More precisely, in our application, heads have always the form "$i(v)$" and each program rule looks like "$i(v)$ *with* v". It is noteworthy to point out that even when we use the same names for constants (building data terms) and truth degrees, the Herbrand Universe of each program and the carrier set of its associated lattice should never be confused, since they are in fact disjoint sets.

A *goal* is a formula built from atomic formulas $B_1, \ldots, B_n$ ($n \geq 0$), truth values of L, conjunctions, disjunctions and aggregations, submitted as a query to the system. In this subset of MALP, the atomic formulas of a *goal* have always the form "$i(P)$", being P a variable symbol. In this way, when running a simple goal like "$i(P)$" (as done in Figure 2), we could obtain several answers meaning something like "when $P = v$, then the resulting truth degree is v", representing all possible interpretations in L for proposition P in the original formula.

The procedural semantics of this subset of the MALP language consists of an operational phase (based on admissible steps that exploits the atoms in the goal), followed by an interpretive phase (that performs arithmetic operations to interpret the resulting formula on the lattice). In the following, $\mathcal{C}[A]$ denotes a formula where A is a sub-expression which occurs in the –possibly empty– context $\mathcal{C}[]$. Moreover, $\mathcal{C}[A/A']$ means the replacement of A by A' in context $\mathcal{C}[]$.

Definition 1 (Admissible Step). *Let $\mathcal{Q}$ be a goal and let σ be a substitution. The pair $\langle \mathcal{Q}; \sigma \rangle$ is a* state. *Given a program $\mathcal{P}$, an* admissible computation *is formalized as a state transition system, whose transition relation $\rightarrow_{AS}$ is defined as the least one satisfying $\langle \mathcal{Q}[A]; \sigma \rangle \quad \rightarrow_{AS} \quad \langle (\mathcal{Q}[A/v])\theta; \sigma\theta \rangle$, where A is the selected atom in $\mathcal{Q}$, $\theta = mgu(\{H = A\})$[4] and "H with v" in $\mathcal{P}$. An* admissible derivation *is a sequence $\langle \mathcal{Q}; id \rangle \rightarrow_{AS} \cdots \rightarrow_{AS} \langle \mathcal{Q}'; \theta \rangle$.*

If we exploit all atoms of a given goal, by applying admissible steps as much as needed during the operational phase, then it becomes a formula with no atoms (a L-expression) which can be then interpreted w.r.t. lattice L as follows.

Definition 2 (Fuzzy Computed Answer). *Let $\mathcal{P}$ be a program, $\mathcal{Q}$ a goal and σ a substitution. Assume that $[\![@]\!]$ is the truth function of connective @ in the lattice $(L, \leq)$ associated to $\mathcal{P}$, such that, for values $r_1, \ldots, r_n, r_{n+1} \in L$, we have that $[\![@]\!](r_1, \ldots, r_n) = r_{n+1}$. Then, we formalize the notion of* interpretive computation *as a state transition system, whose transition relation $\rightarrow_{IS}$ is defined as the least one satisfying: $\langle \mathcal{Q}[@(r_1, \ldots, r_n)]; \sigma \rangle \quad \rightarrow_{IS} \quad \langle \mathcal{Q}[@(r_1, \ldots, r_n)/r_{n+1}]; \sigma \rangle$. An* interpretive derivation *is a sequence $\langle \mathcal{Q}; \sigma \rangle \rightarrow_{IS} \cdots \rightarrow_{IS} \langle \mathcal{Q}'; \sigma \rangle$. When $\mathcal{Q}' = r \in L$, the state $\langle r; \sigma \rangle$ is called a* fuzzy computed answer *(f.c.a.) for that derivation.*

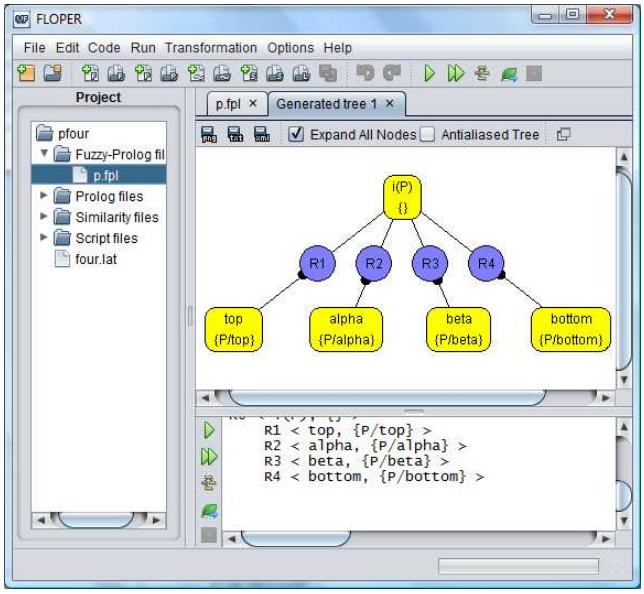


Fig. 2: A work-session with FLOPER solving goal `i(P)`.

[4] Here $mgu(E)$ denotes the *most general unifier* of an equation set E [10].

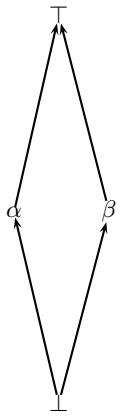
The parser of our FLOPER tool [13] has been implemented by using the Prolog language. Once the application is loaded inside a Prolog interpreter, it shows a menu which includes options for loading/compiling, parsing, listing and saving MALP programs, as well as for executing/debugging fuzzy goals. Moreover, FLOPER has been recently equipped with new options, called "`lat`" and "`show`", for allowing the possibility of respectively changing and displaying the lattice associated to a given program.

A very easy way to model truth-degree lattices for being included into the FLOPER tool is based on the following guidelines. All relevant components of each lattice are encapsulated inside a Prolog file which must necessarily contain the definitions of a minimal set of predicates defining the set of valid elements (`member/1` predicate), the top and bottom elements (`top/1` and `bot/1` predicates), the full or partial ordering established among them (`leq/2` predicate), as well as the repertoire of fuzzy connectives which can be used for their subsequent manipulation. If we have, for instance, some fuzzy connectives of the form $\&_{label_1}$ (conjunction), $|_{label_2}$ (disjunction) or $@_{label_3}$ (aggregation) with arities n_1, n_2 and n_3 respectively, we must provide clauses defining the *connective predicates* "`and_`*label*$_1$`/(n`$_1$`+1)`", "`or_`*label*$_2$`/(n`$_2$`+1)`" and "`agr_`*label*$_3$`/(n`$_3$`+1)`", where the extra argument of each predicate is intended to contain the result achieved after the evaluation of the proper connective. Finally, for the purposes of the current work, we also require for each lattice a Prolog fact of the form `members(L)` being the `L` a list containing the set of truth degrees belonging to the modeled lattice (or at least a representative subset of them when working with infinite lattices) for being used when interpreting propositional variables of fuzzy formulae. For instance, a lattice defining the simplest notion of binary lattice should implement predicate `member/1` with facts `member(0)` and `member(1)` (including also `members([0,1])`) and the Boolean conjunction could be defined by the pair of facts `and_bool(0,_,0)` and `and_bool(1,X,X)`.

Consider now the following partially ordered lattice $\mathcal{F}$ in the diagram of Figure 3, which is equipped with conjunction, disjunction and implication connectives based on the *Gödel* logic described in Figure 1, but with the particularity that now, in the general case, the *Gödel*'s conjunction must be expressed as $\&_{\mathtt{G}}(x, y) \triangleq inf(x, y)$, where it is important to note that we must replace the use of "*min*" by "*inf*" in the connective definition (and similarly for the disjunction connective, where "*max*" must be substituted by "*sup*").

To this end, observe in the Prolog code accompanying the graphic in Figure 3 that we have introduced clauses defining the primitive operators "`pri_inf/3`" and "`pri_sup/3`" which are intended to return the *infimum* and *supremum* of two elements. Related with this fact, we must point out the following aspects:

- Since truth degrees α and β are incomparable, then any call to goals of the form "`?- leq(alpha,beta).`" or "`?- leq(beta,alpha).`" will always fail.
- The goal "`?- pri_inf(alpha,beta,X).`", instead of failing, successfully produces the desired result "`X=bottom`".




```
member(bottom).  member(alpha).
member(beta).    member(top).

members([bottom,alpha,beta,top]).

leq(bottom,X).  leq(alpha,alpha).
leq(beta,beta). leq(X,top).

and_godel(X,Y,Z) :- pri_inf(X,Y,Z).

pri_inf(bottom,X,bottom):-!.
pri_inf(alpha,X,alpha):-leq(alpha,X),!.
pri_inf(beta,X,beta):-leq(beta,X),!.
pri_inf(top,X,X):-!.
pri_inf(X,Y,bottom).
```

Fig. 3: Lattice of truth degrees $\mathcal{F}$ modeled in Prolog.

– Note anyway that the implementation of the "`pri_inf/3`" predicate is mandatory for coding the general definition of "`and_godel/3`" (a similar reasoning follows for "`pri_sup/3`" and "`or_godel/3`").

3 Looking for Models with FuzzyXPath

The subset of the MALP language detailed in Section 2 suffices for developing a simple fuzzy theorem prover, where it is important to remark that our tool can cope with different lattices (not only the real interval [0,1]) containing a finite number of elements -marked in "`members`"- maintaining full or partial ordering among them. Hence, we can use FLOPER for enumerating the whole set of interpretations and models of fuzzy formulae. To this end, only a concrete lattice L is required in order to automatically build a program composed by a set of facts of the form "$i(\alpha)$ *with* α", for each $\alpha \in L$. For instance, the MALP program associated to lattice $\mathcal{F}$ in Figure 3 looks like:

```
i(top)      with   top.
i(alpha)    with   alpha.
i(beta)     with   beta.
i(bottom)   with   bottom.
```

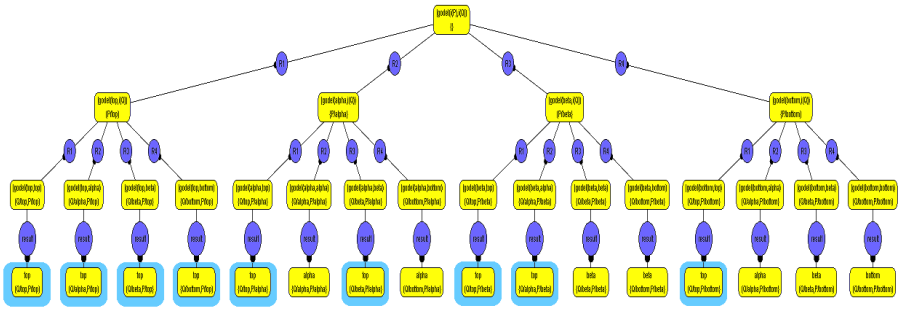


Fig. 4: A work-session with FLOPER solving formula $P \vee Q$ (16 interpretations, 9 models).

Once the lattice and the residual program have been loaded into FLOPER, the formula to be evaluated is introduced as a goal to the system following some conventions:

- If P is a propositional variable in the original formula, then it is denoted as "`i(P)`" in the goal F.
- If & is a conjunction of a certain logic "label" in the original formula, then it is denoted as "&`label`" in goal F.
- For disjunctions, negations and implications, use respectively the patterns "|`label`", "@`no_label`" and "@`im_label`" in F.
- For other aggregators use "@`label`" in F.

In what follows we discuss some examples related with the lattice shown in Figure 3 and its residual MALP program just seen before. Firstly, if we execute goal "`i(P)`" into FLOPER, we obtain the four interpretations for `P` shown in Figure 2. On the other hand, consider now the propositional formula $P \vee Q$, which is translated into the MALP goal "`(i(P) | i(Q))`" and after being executed with FLOPER, the tool returns a tree as seen in Figure 4 whose 16 leaves represent the whole set of interpretations, where 9 of them -inside blue clouds- are models (see part of the corresponding XML file produced by FLOPER in Figure 5). Here, each state contains its corresponding goal and substitution components and they are drawn inside yellow circles. Admissible steps, coloured in blue, are labelled with the program rule they exploit. Finally, those blue circles annotated with word "*is*", correspond to interpretive steps. Sometimes we include blue circles labelled with "*result*" which represents a chained sequence of interpretive steps.

Let us recall now that XPath was designed as a query language for XML text in which the path of the underlying tree of any XML document is used to describe the query (subsequent nodes on XPath expressions are separated by one slash '`/`' or a double slash '`//`', being this last case useful for overriding several nodes). Moreover, XPath expressions can be adorned with Boolean conditions (between

```
<node>
  <rule>R0</rule>
  <goal>or_godel(i(P),i(Q))</goal>
  <substitution>{}</sub>
  <children>
    <node>
      <rule>R1</rule>
      <goal>or_godel(bottom,i(Q))</goal>
      <sub>{P/bottom}</sub>
      <children>
        <node>
          <rule>R1</rule>
          <goal>or_godel(bottom,bottom)</goal>
          <sub>{Q/bottom,P/bottom}
          </sub>
          <children>
            <node>
              <rule>result</rule>
              <goal>bottom</goal>
              <sub>{Q/bottom,P/bottom}
              </sub>
              <children>
              </children>
            </node>
          </children>
        </node>
        ...
```

Fig. 5: Part of the XML file representing the execution tree shown in Figure 4.

square brackets '`[]`') on nodes and leaves to restrict the number of answers of the query. In our fuzzy version of XPath, a FuzzyXPath expression defines, w.r.t. an XML document, a sequence of subtrees of the XML document where each subtree has an associated *retrieval status value, rsv*. XPath conditions, which are defined as fuzzy operators applied to XPath expressions, compute a new rsv from the rsv's of the involved XPath expressions, which at the same time, provides a rsv to the node. We consider three fuzzy versions for each one of the classical conjunction and disjunction operators describing *pessimistic*, *realistic* and *optimistic* scenarios, see Figure 1. In XPath expressions the fuzzy versions of the connectives make harder to hold conditions, and therefore can be used to debilitate/force 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 $(p_1 * r_1 + p_2 * r_2)/p_1 + p_2$.

With our FuzzyXPath tool we have executed "`//node[goal='top']/sub`" against the XML file shown in Figure 5, which was generated by FLOPER when producing the proof tree drawn in Figure 4, thus returning as output the new XML document listed in Figure 6. As illustrated in Figure 5, note that the XML

```
<result>
  <sub rsv=1>{Q/top,P/top}</sub>
  <sub rsv=1>{Q/alpha,P/top}</sub>
  <sub rsv=1>{Q/beta,P/top}</sub>
  <sub rsv=1>{Q/bottom,P/top}</sub>
  <sub rsv=1>{Q/top,P/alpha}</sub>
  <sub rsv=1>{Q/beta,P/alpha}</sub>
  <sub rsv=1>{Q/top,P/beta}</sub>
  <sub rsv=1>{Q/alpha,P/beta}</sub>
  <sub rsv=1>{Q/top,P/bottom}</sub>
</result>
```

Fig. 6: XML file obtained after evaluating an XPath query.

files representing proof trees exported by FLOPER, are always rooted with the `node` label, whose children are based on four kinds of 'tags' (this structure is nested as much as needed):

- `rule`, which indicates the program rule exploited to reach the current node (the virtual rule `R0` is pointed out only in the initial node),
- `goal`, which contains the MALP expression under evaluation, that is, the formula that the system is trying to prove on its current initial/intermediate/final step. Note that, when in our example such value is `top`, then we have found a model, where the values assigned to the propositional symbols of the formula are collected in the following tag...
- `sub`, acronym of "substitution", which accumulates the variable bindings performed along a fuzzy logic derivation (i.e., chain of computational steps along a branch of the execution tree) and whose meaning in our target setting, reveals the way of interpreting the propositions contained on a formula for obtaining its models. See for instance Figure 6, where the nine solutions labeled with this tag and reported in the output XML document, indicate each one the truth values for the propositional variables that satisfy the formula with the maximum truth degree. And finally,
- `children`, which contains the set of underlying nodes of the tree in a nested way.

Consider now the more involved formula $P \wedge Q \rightarrow P \vee Q$ which becomes into the MALP goal "`(i(P) & i(Q)) @impl (i(P) | i(Q))`". When interpreted by FLOPER, the system returns a list of answers having all them the maximum truth degree "*top*", which proves that this formula is a tautology, as wanted. Assume now a more general version with the following shape $F_n = P_1 \wedge \ldots \wedge P_n \rightarrow P_1 \vee \ldots \vee P_n$. With respect to the efficiency of the method presented here, we have studied the behaviour of formula F_n in the table of Figure 7. In the horizontal axis we represent the number n of different propositional variables appearing in the formula, whereas the vertical axis refers to the number of seconds needed to obtain the whole set of interpretations (all them are models in this case)

for the formula. The benchmarks have been performed using a computer with processor Intel Core Duo, with 2 GB RAM and Windows Vista. Both the red and blue lines refers to the method just commented along this paper, but whereas the red line indicates that the derivation tree has been produced by performing admissible and interpretive steps according Definitions 1 and 2, respectively, the blue line refers to the execution of the Prolog code obtained after compiling with FLOPER the MALP program and goal associated to our intended formula.

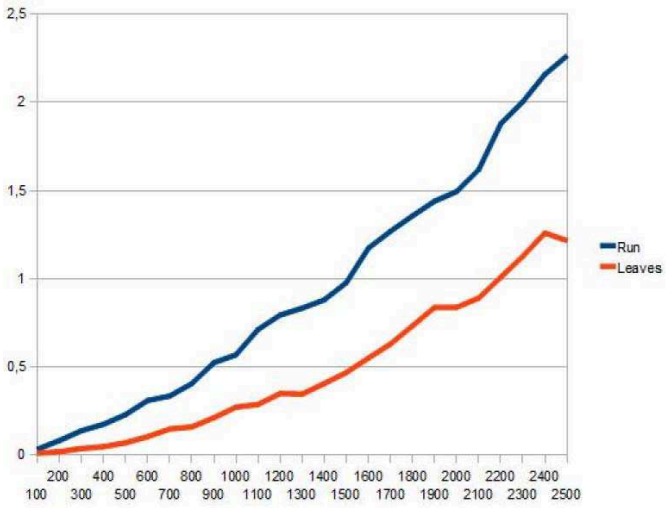


Fig. 7: Efficiency of the method.

The results achieved in Figure 7 show that our method has a nice behaviour in both cases, even for formulae with a big number of propositional variables. Of course, the method does not try to compete with SAT techniques (which are always faster and can deal with more complex formulae containing many more propositional variables), but it is important to remark again that in our case, we face the problem of finding the whole set of models for a given formula, instead of only focusing on satisfiability.

We address now formula F_n because it illustrates one key point of this paper. Note that there are $|L|^n$ interpretations for that formula, where $|L|$ is the cardinality of the carrier set of lattice L that models truth degrees. For our example lattice of Figure 3, with four elements, we have 4^n interpretations. Consider, for example, that we are interested in proving that a certain formula, say F_5, is a tautology. In [9] we would have to search at least one interpretation that is not model of F_5 to prove that it is not a tautology, but since there exist $4^5 = 1024$ interpretations, this task is not suitable to be made by hand. To overcome this problem we use FuzzyXPath to automatically search in the XML file generated by FLOPER. The manual task, then, is reduced to designing the FuzzyXPath

query. In this case, since we are interested in proving that F_5 is a tautology, our FuzzyXPath query should be `//node[rule='result' & goal<>'top']/sub`, that is, the system searches nodes whose `rule` tag contain the text "result" (i.e., we are looking for leaves in the tree) and whose tag `goal` is not "top" (in order to exclude models). If the output of this query is an empty list of nodes, as it actually is, the formula F_5 is proven to be a tautology, as desired.

FuzzyXPath can also be used for determining the satisfiability of a formula. Consider again formula $P \vee Q$ whose set of interpretations are shown in Figure 4. The query `//node[rule='result' & goal<>'top']/sub` seen above, shows that this formula is not a tautology, since its further evaluation returns the non-empty set:

```
<result>
  <sub rsv=1>{Q/alpha,P/alpha}</sub>
  <sub rsv=1>{Q/bottom,P/alpha}</sub>
  <sub rsv=1>{Q/beta,P/beta}</sub>
  <sub rsv=1>{Q/bottom,P/beta}</sub>
  <sub rsv=1>{Q/alpha,P/bottom}</sub>
  <sub rsv=1>{Q/beta,P/bottom}</sub>
  <sub rsv=1>{Q/bottom,P/bottom}</sub>
</result>
```

Consider now the new query (which is almost antagonist to the previous one) `//node[rule='result' & goal='top']/sub`. In this case, if the output is the empty set, the tested formula is a contradiction (i.e., there is no interpretation satisfying it). Otherwise, it is satisfiable. Furthermore, with FuzzyXPath we can come back to the main purpose of [9], that is listing the set of models of a formula instead of just deciding whether it is satisfiable or not. In particular, the query to list the set of models is the one presented for deciding the satisfiability of the formula at the beginning of this paragraph. Observe in Figure 6 the output of this query w.r.t. formula $P \vee Q$.

Until now we have made use of FuzzyXPath to decide immediately the satisfiability or not of a certain formula. With respect to the queries we have presented, we were interested only in whether their answer set were empty or not. Now we present a query which, by making use of the fuzzy capabilities of FuzzyXPath, returns the list of interpretations together with extra information (into the rsv attribute) about the extent in which they satisfy the formula or not. Consider again formula $P \vee Q$, part of whose derivation tree is represented in the form of the XML file provided by FLOPER in Figure 5. This formula is satisfiable but not a tautology, that is, some of its interpretations satisfy it but other ones do not.

Let us focus now on query `//node[rule='result'&(goal='top' avg{3,1}, goal<>'top')]/sub` for such formula. Here, we ask for those states which are leaves of the tree (condition `rule='result'`) and which are either models (condition `goal='top'`) or not (condition `goal<>'top'`), with the particularity that if the leaf is a model, it fulfils the query at a 75% and, if it is not, with a 25%.

The result is the set of interpretations with a *rsv* value (the degree in which they fulfil the query) between 0.75 and 0.25, as shown in the following table:

```
<result>
  <sub rsv=0.75>{Q/top,P/top}</sub>
  <sub rsv=0.75>{Q/alpha,P/top}</sub>
  <sub rsv=0.75>{Q/beta,P/top}</sub>
  <sub rsv=0.75>{Q/bottom,P/top}</sub>
  <sub rsv=0.75>{Q/top,P/alpha}</sub>
  <sub rsv=0.75>{Q/beta,P/alpha}</sub>
  <sub rsv=0.75>{Q/top,P/beta}</sub>
  <sub rsv=0.75>{Q/alpha,P/beta}</sub>
  <sub rsv=0.75>{Q/top,P/bottom}</sub>
  <sub rsv=0.25>{Q/alpha,P/alpha}</sub>
  <sub rsv=0.25>{Q/bottom,P/alpha}</sub>
  <sub rsv=0.25>{Q/beta,P/beta}</sub>
  <sub rsv=0.25>{Q/bottom,P/beta}</sub>
  <sub rsv=0.25>{Q/alpha,P/bottom}</sub>
  <sub rsv=0.25>{Q/beta,P/bottom}</sub>
  <sub rsv=0.25>{Q/bottom,P/bottom}</sub>
</result>
```

This set of answers briefly show the set of interpretations of the formula. For formulas like F_5, whose XML file of 5.5 MB would be impossible to check by hand, this method offers a quick look of the answers, even when they are very numerous.

4 Conclusions and Future Work

In this paper we have recasted from our previous works [9] and [5], two applications developed with our fuzzy logic programming environment FLOPER in order to feedback and reinforce themselves. In the first paper we proposed a technique for evaluating propositional fuzzy formulae in an alternative way than fuzzy SAT/SMT methods, while in the second work we used the FuzzyXPath interpreter for analyzing derivation trees exported by FLOPER in XML format in order to help the analysis of fuzzy logic computations. In the current paper we have applied this last capability of FuzzyXPath focusing exclusively on derivation trees associated to fuzzy formulae developed according the methodology proposed in [9]. As a result, we have presented an automatic technique useful for determining important features of such formulae (tautology, contradiction, etc...) by making use of XPath queries with a fuzzy taste. As future work, we are nowadays introducing fuzzy thresholding techniques in our application for improving the efficiency of the tool.

References

1. Almendros-Jiménez, J., Luna, A., Moreno, G.: A XPath Debugger based on Fuzzy Chance Degrees. In: Herrero, P. (ed.) Proc. of OTM'12 Workshops. pp. 669–672. Springer Verlag, Lectures Notes in Computer Science 7567 (2012)
2. Almendros-Jiménez, J., Luna, A., Moreno, G.: Fuzzy logic programming for implementing a flexible xpath-based query language. Electronic Notes in Theoretical Computer Science 282, 3–18 (2012)
3. Almendros-Jiménez, J., Luna, A., Moreno, G.: 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)
4. Almendros-Jiménez, J., Luna, A., Moreno, G.: Fuzzy xpath through fuzzy logic programming. New Generation Computing 33(2), 173–209 (2015), http://dx.doi.org/10.1007/s00354-015-0201-y
5. Almendros-Jiménez, J., Luna, A., Moreno, G., Vázquez, C.: Analyzing Fuzzy Logic Computations with Fuzzy XPath. In: Åke Fredlund, L., Castro, L.M. (eds.) Proc. of XIII Spanish Conference on Programming and Languages, PROLE'2013, Madrid, Spain, September 18-20. pp. 136–150 ("work in progress" track, extended version to appear in ECEASST). Universidad Complutense de Madrid (ISBN: 978-84-695-8331-9) (2013)
6. Ansótegui, C., Bofill, M., Manyà, F., Villaret, M.: Building automated theorem provers for infinitely-valued logics with satisfiability modulo theory solvers. In: Proceedings of the 42nd IEEE International Symposium on Multiple-Valued Logic, ISMVL 2012. pp. 25–30 (2012)
7. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability Modulo Theories. In: Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 825–885. IOS Press (2009)
8. Berglund, A., Boag, S., Chamberlin, D., Fernandez, M., Kay, M., Robie, J., Siméon, J.: XML path language (XPath) 2.0. W3C (2007)
9. Bofill, M., Moreno, G., Vázquez, C., Villaret, M.: Automatic proving of fuzzy formulae with fuzzy logic programming and smt. In: Åke Fredlund, L., Castro, L.M. (eds.) Proc. of XIII Spanish Conference on Programming and Languages, PROLE'2013, Madrid, Spain, September 18-20. pp. 151–165 ("work in progress" track, extended version to appear in ECEASST). Universidad Complutense de Madrid (ISBN: 978-84-695-8331-9) (2013)
10. Lassez, J.L., Maher, M.J., Marriott, K.: Unification Revisited. In: Minker, J. (ed.) Foundations of Deductive Databases and Logic Programming, pp. 587–625. Morgan Kaufmann, Los Altos (1988)
11. Lloyd, J.: Foundations of Logic Programming. Springer-Verlag, Berlin (1987), second edition
12. Medina, J., Ojeda-Aciego, M., Vojtáš, P.: Similarity-based Unification: a multi-adjoint approach. Fuzzy Sets and Systems, Elsevier 146, 43–62 (2004)
13. Moreno, G., Vázquez, C.: Fuzzy logic programming in action with floper. Journal of Software Engineering and Applications 7, 273–298 (2014)
14. Stickel, M.E.: A prolog technology theorem prover: Implementation by an extended prolog compiler. Journal of Automated reasoning 4(4), 353–380 (1988)
15. Vidal, A., Bou, F., Godo, L.: An smt-based solver for continuous t-norm based logics. In: Proceedings of the 6th International Conference on Scalable Uncertainty Management. Lecture Notes in Computer Science, vol. 7520, pp. 633–640 (2012)