Designing Lattices of Truth Degrees for Fuzzy Logic Programming Environments

Juan Antonio Guerrero, María del Señor Martínez, Ginés Moreno and Carlos Vázquez
Department of Computing Systems
University of Castilla-La Mancha
Albacete (02071), Spain
Email: {Juan.Guerrero,Gines.Moreno,Carlos.Vazquez}@uclm.es
MSenor.Martinez@alu.uclm.es

***Abstract*—During the last two decades, several fuzzy extensions of the pure logic language PROLOG have been developed thus producing modern fuzzy logic languages which manage truth degrees beyond the simpler case of *{true, false}*. Such values are usually collected in lattices whose correct design require to take special care when deciding the top/bottom elements, establishing the ordering relation and so on. In this paper we describe a graphical tool devoted to assist the development of such structures. A crucial and distinguishing feature of the tool relies on its capability for generating code in form of PROLOG clauses which can be directly imported by the fuzzy logic programming environment $\mathcal{FLOPER}$ developed too in our research group.**

I. INTRODUCTION

Among other purposes, research in *Declarative Programming* and *Fuzzy Logic* has traditionally provided languages and programming techniques for AI, soft-computing, and so on. In particular, *Logic Programming* [1] has been widely used for problem solving and knowledge representation in the past, with recognized influences in the field of AI [2], [3]. Nevertheless, traditional logic languages do not incorporate techniques or constructs to explicitly treat with uncertainty and approximate reasoning.

To fulfill this gap, *Fuzzy Logic Programming* has emerged as an interesting and still growing research area trying to agglutinate the efforts for introducing fuzzy logic into logic programming. During the last decades, several fuzzy logic programming systems have been developed [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], where the classical inference mechanism of SLD–resolution is replaced with a fuzzy variant which is able to handle partial truth and to reason with uncertainty. All these languages manage truth degrees beyond the classic bi-valued case, and they usually make use of lattices for collecting the set of truth degrees organized according a given ordering relation. This is the case of *Multi-Adjoint Logic Programming* (MALP in brief) [14], [15], [16], [17], [18], [19], one of the most powerful and promising approaches in the area for which we have designed some program transformation techniques (see some examples of fuzzy fold/unfold and partial evaluation in [20], [21], [22], [17], [23], as well as fuzzy tabulation mechanisms [24], [25], [26]). In this framework, a program can be seen as a set of rules each one annotated with a truth degree and a goal is a query to the system plus a substitution (initially the empty substitution, denoted by id). *Admissible steps* (a generalization of the classical *modus ponens* inference rule) are systematically applied on goals in a similar way to classical resolution steps in pure logic programming, thus returning an state composed by a computed substitution together with an expression where all atoms have been exploited. Next, during the so called interpretive phase (see [17]), this expression is interpreted under a given lattice, hence returning a pair $\langle$*truth degree*; *substitution*$\rangle$ which is the fuzzy counterpart of the classical notion of computed answer used in pure logic programming.

During the last years, our developments regarding the design of the $\mathcal{FLOPER}$ tool (*"Fuzzy LOgic Programming Environment for Research"*, see [27], [28], [29] and visit `http://dectau.uclm.es/floper/`), have been devoted to implant on its core a rule-based, easy representation of lattices representing fuzzy notions of truth degrees. The system offers running/debugging capabilities for managing MALP programs, and has served us for developing several applications in emergent fields like cloud computing, fuzzy SAT/SMT and the semantic web [30], [31], [32], [33], [34], [35], [36]. The expressive power of PROLOG rules (i.e., clauses) for implementing rich versions of lattices of truth degrees in a very easy way, as well as its crucial role in further fuzzy logic computations, revealed us the need for assisting the design of such structures in a graphical way. This task is comfortably/accurately performed by the LatticeMaker tool we have recently developed in our research group, whose detailed description constitutes the main goal of the present work.

While Section II summarizes the main features of MALP and $\mathcal{FLOPER}$, paying special attention to multi-adjoint lattices and their nice representation by using standard PROLOG code, in Section III we describe the LatticeMaker tool specially tailored for helping the graphic design of such lattices. Finally, in Section IV we give our conclusions and propose some some lines of future work.

This work was supported by the EU (FEDER), and the Spanish MINECO Ministry (*Ministerio de Economía y Competitividad*) under grant TIN2013-45732-C4-2-P.

II. The Fuzzy Logic Programming Environment $\mathcal{FLOPER}$

We start this section by summarizing the main features of multi-adjoint logic programming (see [14], [15], [16], [18], [19] for a complete formulation of this framework). We work with a first order language, $\mathcal{L}$, containing variables, constants, function symbols, predicate symbols, and several (arbitrary) connectives to increase language expressiveness: implication connectives ($\leftarrow_1, \leftarrow_2, \ldots$); conjunctive operators (denoted by $\&_1, \&_2, \ldots$), disjunctive operators ($\vee_1, \vee_2, \ldots$), and hybrid operators (usually denoted by $@_1, @_2, \ldots$), all of them are grouped under the name of "aggregators".

Additionally, our language $\mathcal{L}$ contains the values of a multi-adjoint lattice $\langle L, \preceq, \leftarrow_1, \&_1, \ldots, \leftarrow_n, \&_n\rangle$, equipped with a collection of *adjoint pairs* $\langle \leftarrow_i, \&_i\rangle$, where each $\&_i$ is a conjunctor which is intended to the evaluation of *modus ponens* [14].

In general, L may be the carrier of any complete bounded lattice where an L-expression is a well-formed expression composed by values and connectives of L, as well as variable symbols 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 an L-expression not containing variable symbols apart from $x_1, \ldots, x_n$. In particular, we will be mainly concerned with the following classical set of adjoint pairs (conjunctors and implications) in $\langle [0,1], \leq\rangle$, where labels L, G and P mean respectively *Łukasiewicz logic*, *Gödel intuitionistic logic* and *product logic* (which different capabilities for modeling *pessimist*, *optimist* and *realistic scenarios*, respectively):

$$\&_{\texttt{P}}(x,y) \triangleq x * y \qquad \leftarrow_{\texttt{P}}(x,y) \triangleq \min(1, x/y)$$

$$\&_{\texttt{G}}(x,y) \triangleq \min(x,y) \qquad \leftarrow_{\texttt{G}}(x,y) \triangleq \begin{cases} 1 & \text{if } y \leq x \\ x & \text{otherwise} \end{cases}$$

$$\&_{\texttt{L}}(x,y) \triangleq \max(0, x+y-1) \qquad \leftarrow_{\texttt{L}}(x,y) \triangleq \min\{x-y+1, 1\}$$

A *rule* is a formula $H \leftarrow_i \mathcal{B}$, where H is an atomic formula (usually called the *head*) and $\mathcal{B}$ (which is called the *body*) is a formula built from atomic formulas $B_1, \ldots, B_n$ — $n \geq 0$ —, truth values of L, conjunctions, disjunctions and aggregations. A *goal* is a body submitted as a query to the system. Roughly speaking, a multi-adjoint logic program is a set of pairs $\langle \mathcal{R}; \alpha\rangle$ (we often write "$\mathcal{R}$ *with* α"), where $\mathcal{R}$ is a rule and α is a *truth degree* (a value of L) expressing the confidence of a programmer in the truth of rule $\mathcal{R}$. By abuse of language, we sometimes refer a tuple $\langle \mathcal{R}; \alpha\rangle$ as a "rule".

To work with this kind of fuzzy logic programming language we have developed the *Fuzzy LOgic Programming Environment for Research* $\mathcal{FLOPER}$ [27], [28], [29]. The system incorporates a graphical user interface and is able to load fuzzy programs and lattices, as well as to evaluate fuzzy goals and depict their evaluation trees. The parser 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, parsing, listing and saving fuzzy programs, as well as for executing fuzzy goals.

All these actions are based in the translation of the fuzzy code into standard Prolog code. The key point is to extend each atom with an extra argument, called *truth variable* of the form "_TV$_\texttt{i}$", which is intended to contain the truth degree obtained after the subsequent evaluation of the atom. To visualize this translation, consider, for instance, the fuzzy (MALP) program,

```
oc(X) <- s(X) &prod (f(X) @aver w(X)).

s(madrid) with 0.8.      s(tokyo) with 0.9.
f(madrid) with 0.8.      f(tokyo) with 0.7.
w(madrid) with 0.9.      w(tokyo) with 0.6.

s(istambul) with 0.3.     s(baku) with 0.3.
f(istambul) with 0.4.     f(baku) with 0.2.
w(istambul) with 0.8.     w(baku) with 0.5.
```

Then, the first clause in our target program is translated into:

```
oc(X, _TV0) :- s(X, _TV1),
               f(X, _TV2),
               w(X, _TV3),
               agr_aver(_TV2, _TV3, _TV4),
               and_prod(_TV1, _TV4, TV0).
```

Moreover, the second clause in our target program, becomes the pure Prolog fact "`s(madrid,0.8)`" while a fuzzy goal like "`oc(X)`", is translated into the pure Prolog goal: "`oc(X, Truth_degree)`" (note that the last truth degree variable is not anonymous now) for which the Prolog interpreter returns the fuzzy computed answers:

- $[\texttt{Truth\_degree} = 0.68, \texttt{X} = \texttt{madrid}]$,
- $[\texttt{Truth\_degree} = 0.18, \texttt{X} = \texttt{istambul}]$,
- $[\texttt{Truth\_degree} = 0.585, \texttt{X} = \texttt{tokyo}]$
- $[\texttt{Truth\_degree} = 0.105, \texttt{X} = \texttt{baku}]$

The previous set of options suffices for running fuzzy programs (the "`run`" choice also uses the clauses contained in file "num.pl" of Figure 1, which represent the default lattice): 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.

On the other hand, $\mathcal{FLOPER}$ has been equipped with an option called "`lat`" for changing the multi-adjoint lattice associated to a given program.

We have conceived a very easy way to model truth-degree lattices for being included into the $\mathcal{FLOPER}$ tool. All relevant components of each lattice can be encapsulated inside a Prolog file which must necessarily contain the definitions of a minimal set of predicates defining the set of valid elements (including special mentions to the "`top`" and "`bottom`"

```
member(X) :- number(X),0=<X,X=<1.                    members([0,0.2,0.4,0.6,0.8,1]).

bot(0).                          top(1).                          leq(X,Y) :- X=<Y.

and\_luka(X, Y, Z) :- pri_add(X, Y, U1), pri_sub(U1, 1, U2), pri_max(0, U2, Z).
and_godel(X, Y, Z) :- pri_min(X, Y, Z).
and_prod(X, Y, Z)  :- pri_prod(X, Y, Z).

or_luka(X, Y, Z)   :- pri_add(X, Y, U1), pri_min(U1, 1, Z).
or_godel(X, Y, Z)  :- pri_max(X, Y, Z).
or_prod(X, Y, Z)   :- pri_prod(X, Y, U1), pri_add(X, Y, U2), pri_sub(U2, U1, Z).

agr_aver(X, Y, Z)  :- pri_add(X, Y, U), pri_div(U, 2, Z).

pri_add(X, Y, Z)   :- Z is X + Y.          pri_min(X, Y, Z) :- (X=<Y, Z=X; X>Y, Z=Y).
pri_sub(X, Y, Z)   :- Z is X - Y.          pri_max(X, Y, Z) :- (X=<Y, Z=Y; X>Y, Z=X).
pri_prod(X, Y, Z)  :- Z is X * Y.          pri_div(X,Y,Z)   :- Z is X/Y.
```

Figure 1. Multi-adjoint lattice modeling truth degrees in the real interval [0,1] (file "num.pl").

ones), the full or partial ordering established among them, as well as the repertoire of fuzzy connectives which can be used for their subsequent manipulation. In order to simplify our explanation, assume that file "bool.pl" refers to the simplest notion of (a binary) adjoint lattice, thus implementing the following set of predicates:

- `member/1` which is satisfied when being called with a parameter representing a valid truth degree. In the case of finite lattices, it is also recommend to implement `members/1` which returns in one go a list containing the whole set of truth degrees. For instance, in the Boolean case, both predicates can be simply modeled by the PROLOG facts: `member(0).`, `member(1).` and `members([0,1]).`
- `top/1` and `bot/1` obviously answer with the top and bottom elements of the lattice, respectively. Both are implemented into "bool.pl" as `top(1).` and `bot(0).`
- `leq/2` models the ordering relation among all the possible pairs of truth degrees, and obviously it is only satisfied when it is invoked with two elements verifying that the first parameter is equal or smaller than the second one. So, in our example it suffices with including into "bool.pl" the facts: `leq(0,X).` and `leq(X,1).`
- Finally, given some fuzzy connectives of the form $\&_{label_1}$ (conjunction), $\vee_{label_2}$ (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. For instance, in the Boolean case, the following two facts easily model the behaviour of the classical conjunction operation: `and_bool(0,_,0).` `and_bool(1,X,X).`

The reader can easily check that the use of lattice "bool.pl" when working with MALP programs whose rules have the form:

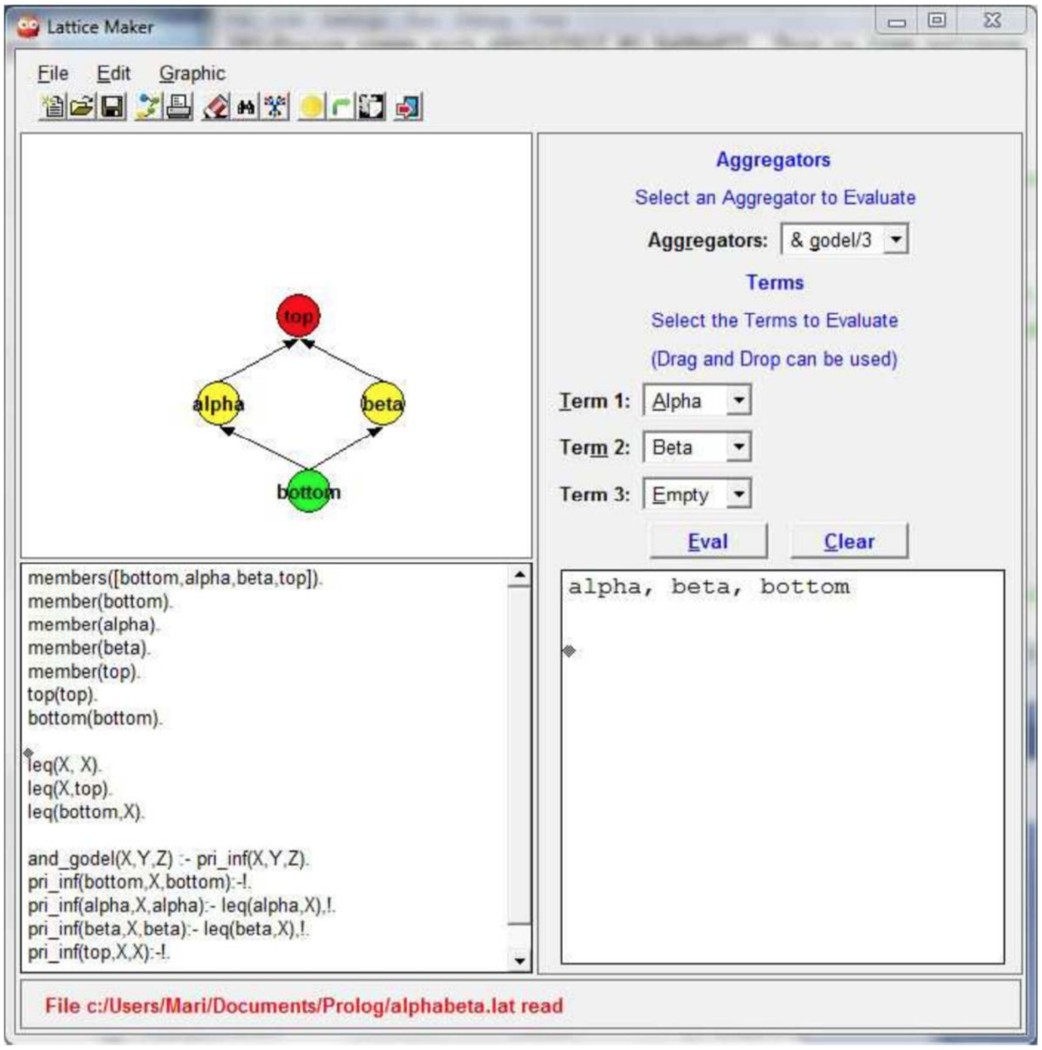
"$A \ \leftarrow_{bool} \ \&_{bool}(B_1, \dots, B_n) \ \ with \ \ 1$"

.... being A and B_i typical atoms[1], successfully mimics the behaviour of classical PROLOG programs where clauses accomplish with the shape "$A \quad : - \quad B_1, \dots, B_n$". As a novelty in the fuzzy setting, when evaluating goals, each output will contain the corresponding PROLOG's substitution (i.e., the *crisp* notion of computed answer obtained by means of classical SLD-resolution) together with the maximum truth degree 1.

On the other hand and following the PROLOG style regulated by the previous guidelines, in file "num.lat" we have included the clauses shown in Figure 1. Here, we have modeled the more flexible lattice (that we will mainly use in our examples, beyond the boolean case) which enables the possibility of working with truth degrees in the infinite space (note that due to this condition in predicate "`members/1`" we include a subset of numbers only) of the real numbers between 0 and 1, allowing too the possibility of using conjunction and disjunction operators recasted from the three typical fuzzy logics proposals described before (i.e., the *Łukasiewicz*, *Gödel* and *product* logics), as well as a useful description for the hybrid aggregator $average$.

Note also that we have included definitions for auxiliary predicates, whose names always begin with the prefix "`pri_`". All of them are intended to describe primitive/arithmetic operators (in our case $+$, $-$, $*$, $/$, min and max) in a PROLOG style, for being appropriately called from the bodies of clauses defining predicates with higher levels of expressiveness (this is the case for instance, of the three kinds of fuzzy connectives we are considering: conjuntions, disjunctions and agreggations).

[1]Here we also assume that several versions of the classical conjunction operation have been implemented with different arities.



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

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

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):-!.
```

```
alpha, beta, bottom
```

File c:/Users/Mari/Documents/Prolog/alphabeta.lat read

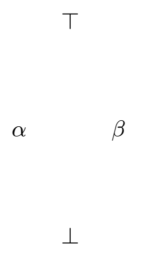
Figure 2. Screenshot of LatticeMaker managing a lattice with four truth degrees.

Since till now we have considered two classical, fully ordered lattices (with a finite and infinite number of elements, collected in files "bool.pl" and "num.pl", respectively), we wish now to introduce a different case coping with a very simple lattice where not always any pair of truth degrees are comparable. So, consider the following partially ordered multi-adjoint lattice in Figure 3 for which the conjunction and implication connectives based on the *Gödel* conform an adjoint pair.... but with the particularity now that, 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.

To this end, observe in the PROLOG code accompanying the figure that we have introduced five clauses defining the new primitive operator "`pri_inf/3`" which is intended to return the *infimum* of two elements. Related with this fact, we must point out the following aspects:

- Note that since truth degrees α and β (or their corresponding representations as PROLOG terms "`alpha`" and "`beta`" used for instance in the definition(s) of "`members(s)/1`") are incomparable then, any call to goals of the form "`?- leq(alpha,beta).`" or "`?- leq(beta,alpha).`" will always fail.
- A goal like "`?- pri_inf(alpha,beta,X).`", or alternatively "`?- pri_inf(beta,alpha,X).`", instead of failing, successfully produces the desired result "`X=bottom`".
- Note anyway that the implementation of the "`pri_inf/1`" predicate is mandatory for coding the general definition of "`and_godel/3`".

Note that $\mathcal{FLOPER}$ does not implement yet any method to graphically show lattices, and precisely is this inability what we try to overcome in the current paper, by presenting a tool for to graphically edit and visualize lattices, in a similar way to what shows Figure 4.



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

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

bot(bottom).      top(top).

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

Figure 3. PROLOG code modeling a partially ordered lattice.

To end this section, we remark that we have provided an instance of the $\mathcal{FLOPER}$ tool in the url `http://dectau.uclm.es/floper/?q=sim/test`. This implementation includes all the functionality of the desktop environment and it is freely available with no further software installation.

III. THE LATTICEMAKER TOOL

The fuzzy logic programming environment $\mathcal{FLOPER}$ can be seen as tool rich enough for coping with MALP programs. Indeed, this environment allows to code programs, to group them into projects, as well as running them and, even, depicting execution trees in a graphical way. However, one thing was left apart of this usability exuberance. Until now, the complex structures known as multi-adjoint lattices were directly coded as PROLOG programs as seen in Section II, with no facility from part of the (in other cases) user-friendly $\mathcal{FLOPER}$ environment. It is evident that this way of designing lattices is tedious and, worst of all, potentially misleading, since the code can be sometimes confusing when dealing with a complex lattice structure.

The solution to this problem is a new software environment called LatticeMaker. This tool is intended to ease the manipulation and design of multi-adjoint lattices, that define the notion of truth degrees for MALP programs.

Usability was one of the major concerns in the design of LatticeMaker. Note that, in the current context of software usage, there are a wide range of users with very different technical abilities, which do not care about the internal features of programs, but their facility of use. The notion of usability of a program is understood as the speed and easiness with which a user performs his tasks using a certain software product. For Jacob Nielsen, pioneer in the diffusion of usability, it is a multidimensional term [37], [38], and, together with Rolf Molich [39], they identify the following set of heuristics for the design of usable software:

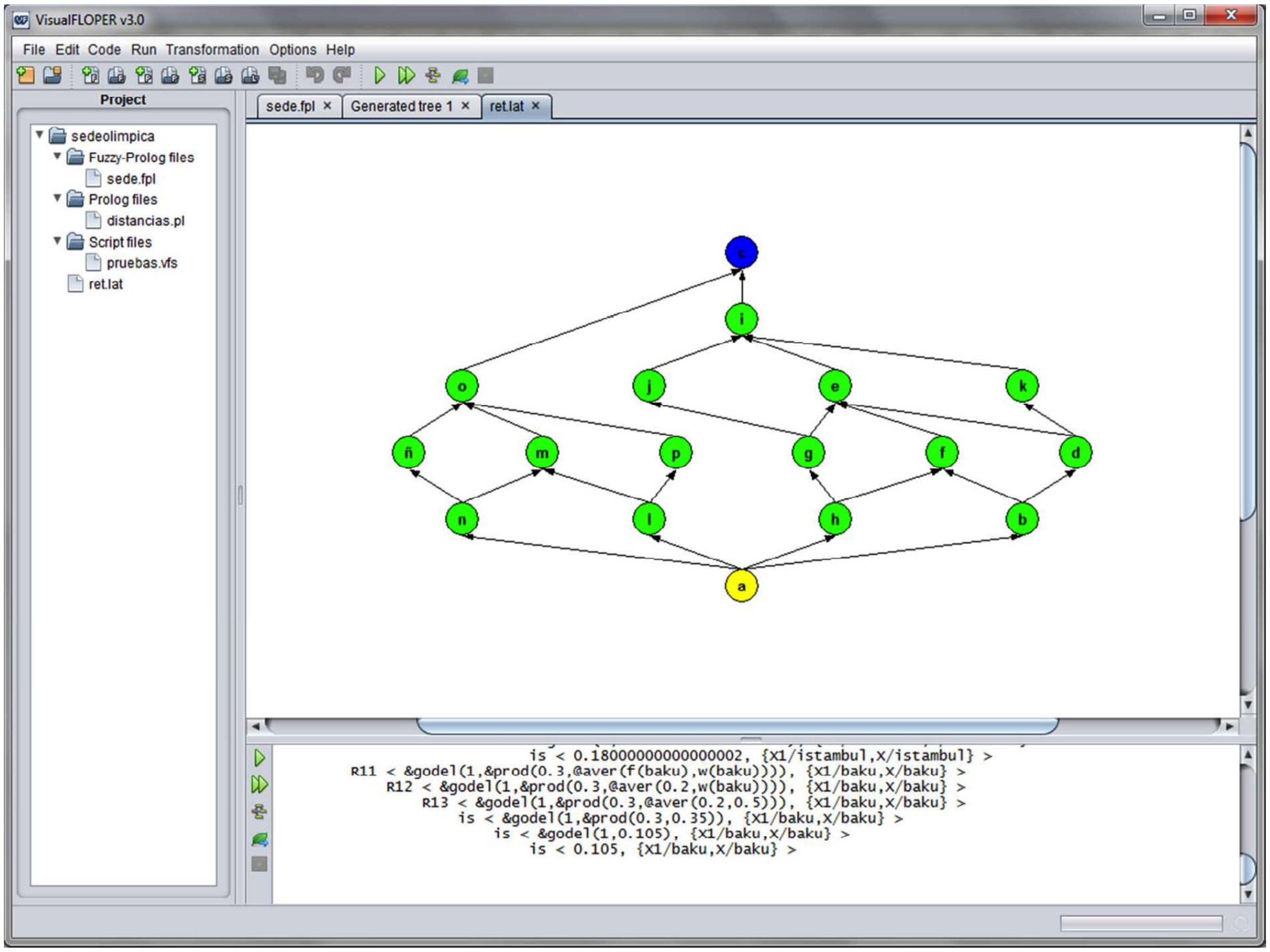
- Visibility of the state of the system: the system has to make visible its state to the user through the corresponding feedback, so they can know what happens or is going to happen.
- Equivalence between the system and the real world: the system has to speak the language of the user, by means of the use of either concepts or expressions that are familiar to them, or showing information in a logic way similarly to the real world.
- Freedom and control for the user: frequently, users mistakenly choose an option, so they need to see clearly an emergency exit. Other times, options of Undo and Redo are necessary.
- Consistency and use of standards: occasionally, users do not know that different words, actions or icons have the same meaning. In these cases it is appropriate to make use of standards.
- Error prevention: even better than a good error message is the prevention of problems by, for instance, informing the user that the action they are performing may fail.
- Recognition instead of memorization: this means less effort by the user. Most frequently used functions must be easily accessible.
- Flexibility and efficiency of use: shortcuts increase speed of use for expert users.
- Minimalist and aesthetic design: dialogs must not contain unusual or irrelevant information so they do not visually compete with the really useful information.
- Diagnosis and error recovery: errors must be expressed in ordinary language (with no code), noticing the problem and, if possible, suggesting some solution.
- Help and documentation: documentation must be focused on tasks, with short lists of concrete steps.

LatticeMaker (see again Figure 2) has entirely been developed in PROLOG (particularly, SWI-PROLOG) through the XPCE library. XPCE/PROLOG is a hybrid environment that merges logic programming with object oriented programming in order to obtain Graphical User Interfaces (GUI's). It was developed as "PCE project" in 1985 by Anjo Abjewierden with the aim of creating an high level environment for C-PROLOG. PCE migrated to X-windows and became compatible with Windows and Unix under the name XPCE. Currently, XPCE is distributed as an integrated package in SWI-PROLOG under GNU license.

Furthermore, this library includes an event handler, graphical controls and user-defined classes. One of the advantages of XPCE is the readability of the resulting programs, its efficiency and the fact that is directly interpretable in a PROLOG environment so its integration with other systems is easy.

Our tool was created to cope with the following set of goals:

- The main purpose was the creation of a GUI for the edition of multi-adjoint lattices associated to fuzzy (MALP)



```
                    is < 0.18000000000000002, {X1/istambul,X/istambul} >
      R11 < &godel(1,&prod(0.3,@aver(f(baku),w(baku)))), {X1/baku,X/baku} >
          R12 < &godel(1,&prod(0.3,@aver(0.2,w(baku)))), {X1/baku,X/baku} >
              R13 < &godel(1,&prod(0.3,@aver(0.2,0.5))), {X1/baku,X/baku} >
                  is < &godel(1,&prod(0.3,0.35)), {X1/baku,X/baku} >
                      is < &godel(1,0.105), {X1/baku,X/baku} >
                          is < 0.105, {X1/baku,X/baku} >
```

Figure 4. Complex lattice as would be depicted in $\mathcal{FLOPER}$ incorporating LatticeMaker.

programs. It is very important that it can cope with the format of lattices defined by $\mathcal{FLOPER}$.

- The lattice would be represented in a graphical way, so the predicates defining it must be interpreted as a graph.
- The edition and modification of the lattice should be also graphical, so the graph could be interactive.
- Another possibility must be the edition of the lattice in text format for the manipulation of the connectives associated to the lattice.
- The result of the edition must be saved in the same format (i.e., PROLOG code) defined by $\mathcal{FLOPER}$.
- The resulting lattice should be also exportable to different formats (XML and image).
- The system must be able to evaluate the behavior of the connectives for the truth degrees defined in the lattice.

LatticeMaker is devised to be embedded into the $\mathcal{FLOPER}$ tool. Therefore, the user would be able to manage all the parts of a given fuzzy application (program and lattice) in a visual, graphical, user-friendly way. This means an incredibly enhancement on readability of the lattices, as we illustrate in Figure 4, where we show a (complex but by no means exaggerate) lattice with the combined use of LatticeMaker and $\mathcal{FLOPER}$.

It is also interesting to note that LatticeMaker is able to work not only with multi-adjoint lattices but with lattices much more general (particularly, complete lattices). Furthermore, since it is nowadays an independent tool, it is also useful for other areas, like mathematics, AI, networks, etc.

LatticeMaker provides a rich set of options and buttons to model multi-adjoint lattices. Options are grouped into four categories, as we are going to describe now:

A. File submenu

This submenu offers options for creating new lattices, loading previously created lattices (as PROLOG programs), printing the lattice in use, saving the lattice and exporting it as a JPG, GIF or XPM image, as a PDF file (through the "print" choice) or as an XML file, as seen in the following code, that represents part of the XML file modeling the lattice "four" (see Figures 2 and 3).

Figure 5. Toolbar of the Edit menu.

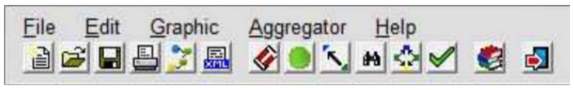


Figure 6. Toolbar of the File menu and graphical area.

```
<?xml version="1.0" encoding="utf-8" ?>

<LATTICE>
  <MEMBERS>
    <MEMBER>top</MEMBER>
    <MEMBER>alpha</MEMBER>
    <MEMBER>beta</MEMBER>
    <MEMBER>bottom</MEMBER>
  </MEMBERS>
  <BOTTOM>bottom</BOTTOM>
  <TOP>top</TOP>
  <ARC>
    <FROM>alpha</FROM>
    <TO>top</TO>
  </ARC>
  ...
</LATTICE>
```

B. Edit submenu

This submenu allows to work in the text area where the system prints the PROLOG clauses defining the current lattice. By default it is set in only-read mode, although the user can easily switch this mode to perform updates. It includes an option for adding a new connective to the lattice. It is also possible to *redraw the graph* for representing the lattice associated to the text area of the program, in order to make visible the possible modifications performed there by the user.

The text area of the editor is sensible to all the usual key combinations: text selection, copy-paste-cut selected text, find, replace and undo last change. It also includes a shortcut to the most common options, illustrated in Figure 5.

C. Graphic submenu

This submenu provides options related with the canvas where the lattice is graphically drawn. These options appear in Figure 6. This is the most intuitive part of the system and users are invited to work directly in the drawn lattice, instead of directly manipulating PROLOG clauses. With right click on the visual elements of the lattice, a menu pops up offering related actions. In particular, right click on a node shows options for deleting and renaming it, and right click over an arc allows to delete the arc. Also, right click over an empty area of the canvas allows to create a new node and to name it. It is possible to move nodes dragging them with the mouse. If the node is dragged to one of the "Term i" controls, it is loaded as the "i-th" parameter of a connective enabling its further evaluation (as we will see afterwards).

In order to create an arc between nodes, the user selects the *Arrow mode*, and clicks over the initial and final nodes, thus creating a link corresponding to the predicate `leq(initial, final).`, where *initial* is the first node clicked and *final* is the last one. Some consideration must be done with this respect:

- Connections with the *top* element as origin are not allowed, and neither are those ones ending in the *bottom* element.
- Connections from one node to itself are not allowed, although it is understood that all nodes fulfils the property of being less or equal to themselves.
- A connection between two nodes is not allowed if the opposite is already stated.

Notably, nodes in the canvas are tagged with the name of the truth degree they are representing. They are coloured following the next conventions, as depicted in Figure 7:

- Red: Not connected nodes.
- Green: Fully connected nodes.
- Purple: Nodes connected only to the top element.
- Orange: Nodes connected only to the bottom element.
- Blue: Top element.
- Yellow: Bottom element.

This submenu also allows to complete the graph, that is, to include the links needed so it can be a complete lattice (link each node so there is a path from the bottom to it, and from it to the top). Finally, through the "normalize" option, the system perform a series of tasks devoted to produce a coherent lattice. It, particularly:

- removes cycles,
- deletes transitional arcs, as seen in Figure 8,
- redraws the lattice in layers so it is easier to understand, as seen in Figure 7, and
- rewrites the PROLOG code according to the represented graph (lattice), among other internal actions.

In order to perform this normalizing process, consider that the relations defined by predicate `leq/2` can be very general and, if expanded, it might generate many redundant connexions. With the aim of removing the unnecessary arcs, a new predicate called `arc/2` is defined. It is devoted to identify only the arcs to be displayed in the graph. From then a procedure to translate the set of `leq/2` predicates to a new set of `arc/2` predicates is defined so both ones can coexist in the same lattice. That is, LatticeMaker transforms the loaded lattice so the ordering relation is defined in terms of both the `leq/2` predicate and the `arc/2` predicate, and it is done in such a way that the clauses defining `leq/2`, are redefined in order to invoke the other ones and are always the same in every lattice, concretely:

```
leq(X, X).
leq(X, Y):- arc(X, Z), leq(Z, Y).
```

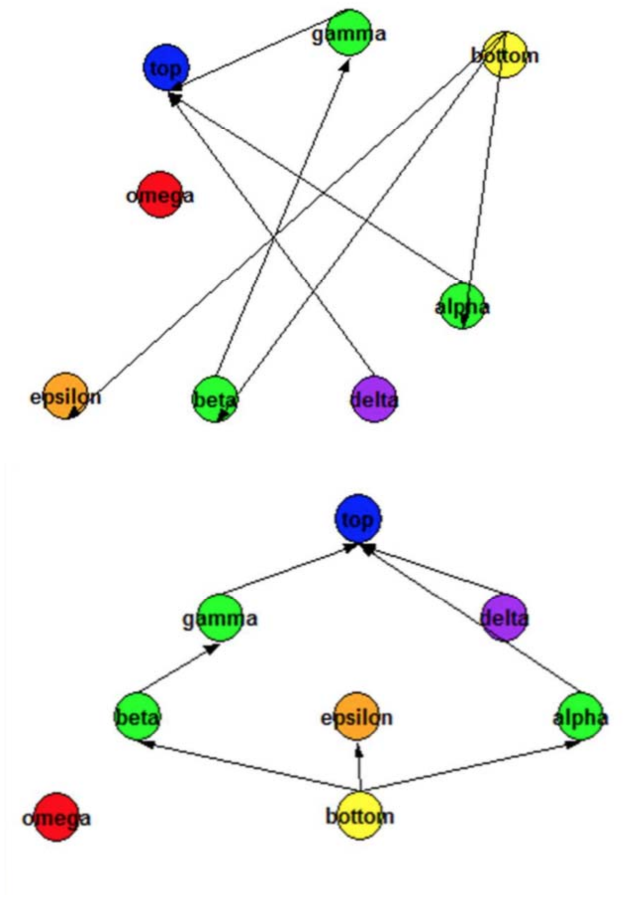


Figure 7. A bad-formed “lattice” exhibiting all the colors in our convention, where the upper one needs to be normalized and the one below is already normalized and displayed by layers.

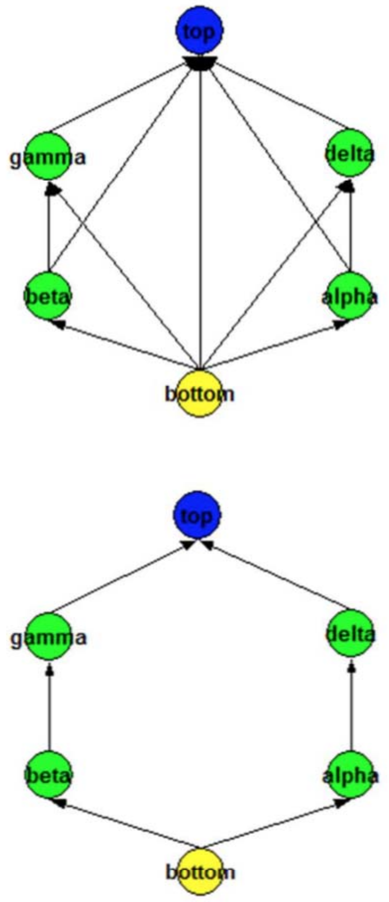


Figure 8. A lattice before and after removing redundant arcs.

Observe that the first clause copes with the idempotent property, that states that each element is less or equal to itself. The second clause makes use of the `arc/2` predicate to build the notion of “less or equal”. Then, an element is less or equal to another if there is an arc connecting it to a third element that is less or equal to the second one. By executing the second clause the first clause is eventually reached, or the system runs out of “arcs” to perform the second clause and the predicate fails (that is, the relation does not hold). The use of predicate `arc/2` allows to easily avoid cycles and transitivity relations, resulting in a more readable lattice definition. The resulting lattice is saved with the name of the original one preceded by prefix “generated_”.

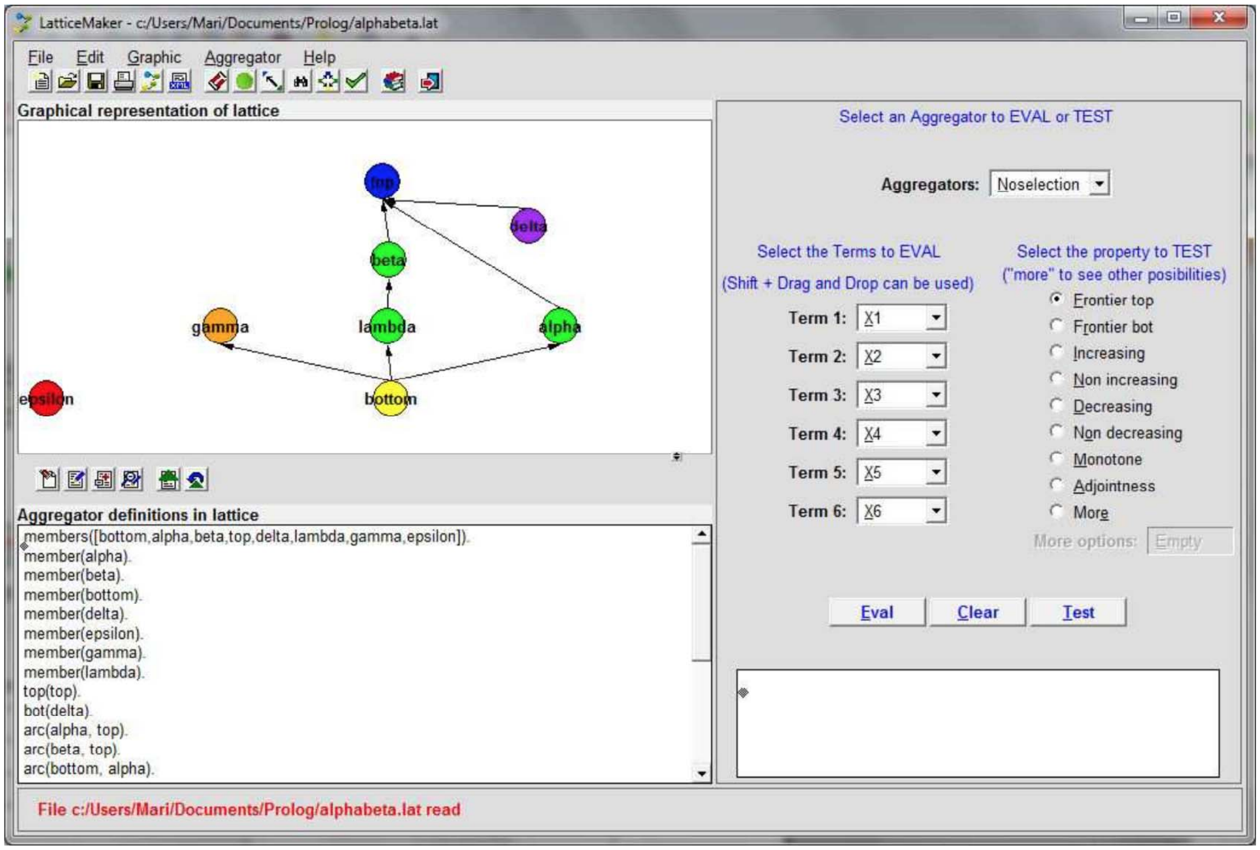
With respect to the representation of the lattice by horizontal layers, the system begins representing the top element, centered on its own layer. Then, it depicts in the layer immediately below the nodes directly connected with top, distributing the space in the layer equally between the nodes. It also takes into account the nodes not reachable from top but from bottom. The system continues then adding each time a new layer with the subsequent nodes, distributing the space, until reaching the bottom element. Finally, it draws the disconnected nodes. While drawing all nodes, their colours are chosen according to the way they are connected (or not) with top and bottom, following the convention previously described.

D. *Aggregator submenu*

As seen in Figure 2, the right area of LatticeMaker is occupied by the “Aggregator submenu”, that is the part of the tool adressing tasks related with the definitions of connectives. All its options require the user to select some of the aggregators of the lattice in the popup menu labelled as “Aggregator”. Such popup takes its values from the connectives defined in the loaded lattice, so, previously, it is mandatory to load some lattice.

Every aggregator has an associated arity that determines the number of parameters it works with. Once an aggregator is selected, the system unlocks as much controls labelled as “Term i” as its arity.

The “Term i” controls are popup menus that take their values from the truth values defined by the lattice, together with an “undefined” value, called “xi”, which acts as a variable in the evaluation of the connectives.


LatticeMaker - c:/Users/Mari/Documents/Prolog/alphabeta.lat
File
Edit
Graphic
Aggregator
Help
Graphical representation of lattice
top
delta
beta
gamma
lambda
alpha
epsilon
bottom
Aggregator definitions in lattice
Select an Aggregator to EVAL or TEST
Select the Terms to EVAL
(Shift + Drag and Drop can be used)
Select the property to TEST
("more" to see other posibilities)
Frontier top
Frontier bot
Increasing
Non increasing
Decreasing
Non decreasing
Monotone
Adjointness
More
More options:
Eval
Clear
Test

Aggregators: Noselection

Term 1:	X1
Term 2:	X2
Term 3:	X3
Term 4:	X4
Term 5:	X5
Term 6:	X6

[✓ Frontier top] Frontier bot Increasing Non increasing Decreasing Non decreasing Monotone Adjointness More

More options: Empty

```
members([bottom,alpha,beta,top,delta,lambda,gamma,epsilon]).
member(alpha).
member(beta).
member(bottom).
member(delta).
member(epsilon).
member(gamma).
member(lambda).
top(top).
bot(delta).
arc(alpha, top).
arc(beta, top).
arc(bottom, alpha).
```

File c:/Users/Mari/Documents/Prolog/alphabeta.lat read

Figure 9. Screenshot of LatticeMaker managing a non well-formed lattice.

This submenu offers the following options:

- Evaluate: It allows to evaluate the connective with the values selected on each "Term i" popup as its parameters. The result of the evaluation is shown in the box below.
- Test aggregator: To use this option the user has to select a property to be tested in the aggregator.We are nowadays implementing a wide range of properties into the system.

In Figure 9 we show the current look of LatticeMaker. In this case we have loaded a badly formulated lattice with an incomplete ordering relation.

IV. Conclusions and Future Work

$\mathcal{FLOPER}$ is a "Fuzzy LOgic Programming Environment for Research" trying to help the development of applications supporting approximated reasoning and uncertain knowledge in the fields of AI, symbolic computation, soft-computing, semantic web, declarative programming and so on. The tool, which is able to directly translate a powerful kind of fuzzy logic programs belonging to the so-called "multi-adjoint logic approach" into standard Prolog code, currently offers running/debugging/tracing capabilities with close connections to other sophisticated manipulation techniques (program optimization, program specialization, etc.) under development in our research group. Our philosophy is to friendly connect this fuzzy framework with Prolog programmers: our system, apart for being implemented in Prolog, also translates the fuzzy code to classical clauses (in two different representations) and, as we have seen in this paper, a wide range of lattices modeling powerful and flexible notions of truth degrees also admit a nice rule-based characterization into Prolog.

The main goal of this work has been the introduction of a graphical tool for aiding the construction of such structures whose comfortable and accurate design has crucial importance for further fuzzy logic computations. It is important to remark that LatticeMaker has not been conceived for competing with other commercial tools developed for academic or industrial purposes (which must be even more popular or powerful) since all them lack of the ability for generating Prolog code modeling lattices according the guidelines required by $\mathcal{FLOPER}$. Anyway, apart for using LatticeMaker when feeding the $\mathcal{FLOPER}$ system with lattices of truth-degrees, the tool can be used in isolation for alternative mathematical, academic or research purposes. Since in the current version of the applicacion we have mainly focused on the treatment of the elements and the ordering relation established on a given lattice, for the future we plan to reinforce the design of wide repertoires of connectives and to check their properties in a graphical aided way.

References

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

[2] I. Bratko, *Prolog Programming for Artificial Intelligence*. Addison Wesley, 2000.

[3] J. Lloyd, "Declarative programming for artificial intelligence applications," *SIGPLAN Not.*, vol. 42, no. 9, pp. 123–124, 2007.

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

[5] S. Guadarrama, S. Muñoz, and C. Vaucheret, "Fuzzy Prolog: A new approach using soft constraints propagation," *Fuzzy Sets and Systems*, vol. 144, no. 1, pp. 127–150, 2004.

[6] M. Ishizuka and N. Kanai, "Prolog-ELF Incorporating Fuzzy Logic," in *Proceedings of the 9th International Joint Conference on Artificial Intelligence, IJCAI'85*, A. K. Joshi, Ed. Morgan Kaufmann, 1985, pp. 701–703. [Online]. Available: http://dl.acm.org/citation.cfm?id=1623611.1623612

[7] D. Li and D. Liu, *A fuzzy Prolog database system*. John Wiley & Sons, Inc., 1990.

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

[9] P. Vojtáš, "Fuzzy Logic Programming," *Fuzzy Sets and Systems*, vol. 124, no. 1, pp. 361–370, 2001.

[10] P. Vojtáš and L. Paulík, "Query answering in normal logic programs under uncertainty," in *Proc. of 8th. European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty (ECSQARU-05), Barcelona, Spain*, L. Godó, Ed. Lecture Notes in Computer Science 3571, Springer Verlag, 2005, pp. 687–700.

[11] U. Straccia, "Managing uncertainty and vagueness in description logics, logic programs and description logic programs," in *Reasoning Web, 4th International Summer School, Tutorial Lectures*, ser. Lecture Notes in Computer Science, no. 5224. Springer Verlag, 2008, pp. 54–103.

[12] P. Julián, C. Rubio, and J. Gallardo, "Bousi∼prolog: a prolog extension language for flexible query answering," *Electronic Notes in Theoretical Computer Science*, vol. 248, pp. 131–147, 2009. [Online]. Available: http://dx.doi.org/10.1016/j.entcs.2009.07.064

[13] C. Rubio-Manzano and P. Julián-Iranzo, "A fuzzy linguistic prolog and its applications," *Journal of Intelligent and Fuzzy Systems*, vol. 26, no. 3, pp. 1503–1516, 2014. [Online]. Available: http://dx.doi.org/10.3233/IFS-130834

[14] J. Medina, M. Ojeda-Aciego, and P. Vojtáš, "Similarity-based Unification: a multi-adjoint approach," *Fuzzy Sets and Systems*, vol. 146, pp. 43–62, 2004. [Online]. Available: http://dblp.uni-trier.de/db/journals/fss/fss146.html#MedinaOV04

[15] ——, "Multi-adjoint logic programming with continuous semantics," *Proc. of Logic Programming and Non-Monotonic Reasoning, LP-NMR'01, Springer-Verlag, LNAI*, vol. 2173, pp. 351–364, 2001.

[16] ——, "A procedural semantics for multi-adjoint logic programing," *Progress in Artificial Intelligence, EPIA'01, Springer-Verlag, LNAI*, vol. 2258, no. 1, pp. 290–297, 2001.

[17] P. Julián, G. Moreno, and J. Penabad, "Operational/Interpretive Unfolding of Multi-adjoint Logic Programs," *Journal of Universal Computer Science*, vol. 12, no. 11, pp. 1679–1699, 2006.

[18] P. Julián, G. Moreno, and J. Penabad, "On the declarative semantics of multi-adjoint logic programs," in *Bio-Inspired Systems: Computational and Ambient Intelligence, 10th International Work-Conference on Artificial Neural Networks, IWANN'09, Salamanca, Spain, June 10-12, 2009, Proceedings, Part I*, ser. Lecture Notes in Computer Science, vol. 5517. Springer, 2009, pp. 253–260.

[19] P. Morcillo, G. Moreno, J. Penabad, and C. Vázquez, "Dedekind-MacNeille completion and cartesian product of multi-adjoint lattices," *International Journal of Computer Mathematics*, vol. 89, no. 13-14, pp. 1742–1752, 2012. [Online]. Available: http://dx.doi.org/10.1080/00207160.2012.689826

[20] P. Julián, G. Moreno, and J. Penabad, "On Fuzzy Unfolding. A Multi-adjoint Approach," *Fuzzy Sets and Systems*, vol. 154, pp. 16–33, 2005.

[21] G. Moreno, "Building a Fuzzy Transformation System," in *Proc. of the 32nd Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM'2006. Merin, Czech Republic, January 21-27*. Springer Verlag, LNCS 3831, 2006, pp. 409–418.

[22] J. Guerrero and G. Moreno, "Optimizing fuzzy logic programs by unfolding, aggregation and folding," *Electronic Notes in Theoretical Computer Science*, vol. 219, pp. 19–34, 2008.

[23] P. Julián, G. Moreno, and J. Penabad, "An Improved Reductant Calculus using Fuzzy Partial Evaluation Techniques," *Fuzzy Sets and Systems*, vol. 160, pp. 162–181, 2009.

[24] P. Julián, J. Medina, G. Moreno, and M. Ojeda, "Thresholded tabulation in a fuzzy logic setting," *Electronic Notes in Theoretical Computer Science*, vol. 248, pp. 115–130, 2009.

[25] ——, "Efficient thresholded tabulation for fuzzy query answering," *Studies in Fuzziness and Soft Computing (Foundations of Reasoning under Uncertainty)*, vol. 249, pp. 125–141, 2010.

[26] P. Julián, J. Medina, P. Morcillo, G. Moreno, and M. Ojeda-Aciego, "An unfolding-based preprocess for reinforcing thresholds in fuzzy tabulation," in *Advances in Computational Intelligence - Proc of the 12th International Work-Conference on Artificial Neural Networks, IWANN'13*. Springer Verlag, LNCS 7902, PART I, 2013, pp. 647–655.

[27] P. Morcillo, G. Moreno, J. Penabad, and C. Vázquez, "A Practical Management of Fuzzy Truth Degrees using FLOPER," in *Proc. of 4nd International Symposium on Rule Interchange and Applications, RuleML'10*. Springer Verlag, LNCS 6403, 2010, pp. 20–34.

[28] G. Moreno and C. Vázquez, "Fuzzy logic programming in action with FLOPER," *Journal of Software Engineering and Applications*, vol. 7, pp. 237–298, 2014. [Online]. Available: http://dx.doi.org/10.4236/jsea.2014.74028

[29] P. Julián, G. Moreno, J. Penabad, and C. Vázquez, "A fuzzy logic programming environment for managing similarity and truth degrees," *Electronic Proceedings in Theoretical Computer Science*, vol. 173, pp. 71–86, 2014.

[30] C. Vázquez, L. Tomás, G. Moreno, and J. Tordsson, "A fuzzy approach to cloud admission control for safe overbooking," in *Proc. of 10th International Workshop on Fuzzy Logic and Applications, WILF 2013*. Springer Verlag, LNAI 8256, 2013, pp. 212–225.

[31] C. Vázquez, G. Moreno, L. Tomás, and J. Tordsson, "A cloud scheduler assisted by a fuzzy affinity-aware engine," in *IEEE International Conference on Fuzzy Systems, FUZZ-IEEE 2015, Istanbul, Turkey, August 2-5*, 2015, p. 8 (in press).

[32] M. Bofill, G. Moreno, C. Vázquez, and M. Villaret, "Automatic proving of fuzzy formulae with fuzzy logic programming and SMT," in *Proc. of XIII Spanish Conference on Programming and Languages, PROLE'2013, Madrid, Spain, September 18-20*, L. Fredlund, Ed. ECE-ASST (extended version in press), 2013, pp. 151–165.

[33] J. M. Almendros-Jiménez, M. Bofill, A. Luna, G. Moreno, C. Vázquez, and M. Villaret, "Fuzzy xpath for the automatic search of fuzzy formulae models," in *Scalable Uncertainty Management: Proceedings of the 9th International Conference SUM 2015, Quebec, Canada, September 16-18*. Springer Verlag, LNAI 9310, 2015, pp. 385–398.

[34] J. M. Almendros-Jiménez, A. Luna, and G. Moreno, "Fuzzy logic programming for implementing a flexible xpath-based query language," *Electronic Notes in Theoretical Computer Science, Elsevier*, vol. 282, pp. 3–18, 2012.

[35] ——, "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*. Springer Verlag, LNCS 7903, Part II, 2013, pp. 300–311.

[36] ——, "Fuzzy xpath through fuzzy logic programming," *New Generation Computing*, vol. 33, no. 2, pp. 173–209, 2015. [Online]. Available: http://dx.doi.org/10.1007/s00354-015-0201-y

[37] J. Nielsen, "A virtual protocol model for computer-human interaction," *International Journal of Man-Machine Studies*, vol. 24, no. 3, pp. 301–312, 1986. [Online]. Available: http://dx.doi.org/10.1016/S0020-7373(86)80028-1

[38] ——, "Enhancing the explanatory power of usability heuristics," in *Conference on Human Factors in Computing Systems, CHI 1994, Boston, Massachusetts, USA, April 24-28, 1994, Proceedings*, B. Adelson, S. T. Dumais, and J. S. Olson, Eds. ACM, 1994, pp. 152–158. [Online]. Available: http://doi.acm.org/10.1145/191666.191729

[39] R. Molich and J. Nielsen, "Improving a human-computer dialogue," *Commun. ACM*, vol. 33, no. 3, pp. 338–348, 1990. [Online]. Available: http://doi.acm.org/10.1145/77481.77486