

# Automatic Proving of Fuzzy Formulae with Fuzzy Logic Programming and SMT

Miquel Bofill<sup>1</sup>, Ginés Moreno<sup>2</sup>, Carlos Vázquez<sup>2</sup> and Mateu Villaret<sup>1</sup>

<sup>1</sup> [Miquel.Bofill@udg.edu](mailto:Miquel.Bofill@udg.edu), [Mateu.Villaret@udg.edu](mailto:Mateu.Villaret@udg.edu)

Department of Computer Science, Applied Mathematics and Statistics  
University of Girona  
17071 Girona (Spain)

<sup>2</sup> [Gines.Moreno@uclm.es](mailto:Gines.Moreno@uclm.es), [Carlos.Vazquez@uclm.es](mailto:Carlos.Vazquez@uclm.es)

Department of Computing Systems  
University of Castilla-La Mancha  
02071 Albacete (Spain)

**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*. We firstly recall an SMT (*Satisfiability Modulo Theories*) based method for automatically proving theorems in relevant infinitely-valued (including *Lukasiewicz* and *Gödel*) logics. Next, instead of focusing on satisfiability (i.e., proving the existence of at least one model) or unsatisfiability, our interest moves to the problem of finding the whole set of models (with a finite domain) for a given fuzzy formula. We propose an alternative method based on fuzzy logic programming where the formula is conceived as a goal whose derivation tree contains on its leaves all the models of the original formula, by exhaustively interpreting each propositional symbol in all the possible forms according the whole set of values collected on the underlying lattice of truth-degrees.

**Keywords:** Fuzzy Logic Programming; Automatic Theorem Proving; SMT

## 1 Introduction

Research on SAT (Boolean Satisfiability) and SMT (Satisfiability Modulo Theories) [BSST09] 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 [ABMV12, VBG12]. Moreover, if automatic theorem solving supposes too an starting point for the foundations of logic programming as well as one of its important application fields [Llo87, Sti88, Fit96, Apt90, Bra00], in this work we will show some preliminary guidelines about how fuzzy logic programming can face the automatic proving of fuzzy theorems.

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  $F : (In' \wedge Out) \vee (In \wedge Out')$ . Depending on how we interpret

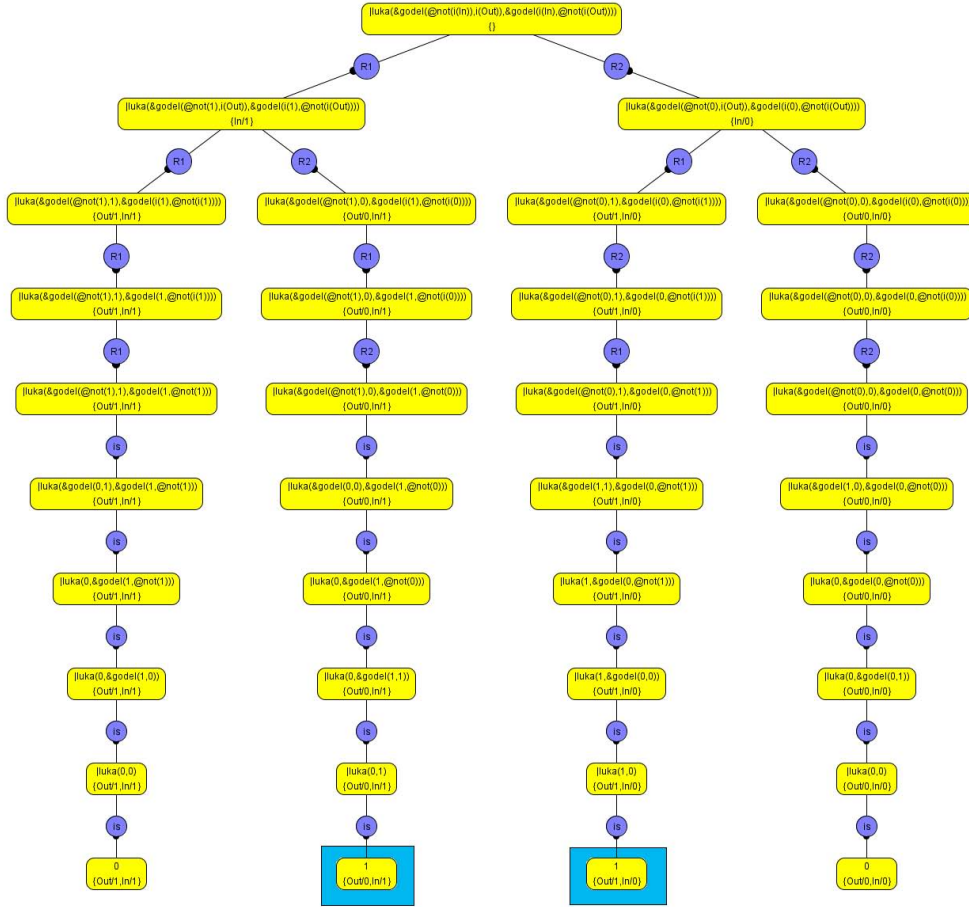


Figure 1: Interpreting a formula with two values for signals/propositions *In* and *Out*.

each propositional symbol, we obtain the following final set of interpretations for the whole formula:

$$\begin{array}{ll}
 I1: \{In = 0, Out = 0\} \Rightarrow F = 0 & I2: \{In = 0, Out = 1\} \Rightarrow F = 1 \\
 I3: \{In = 1, Out = 0\} \Rightarrow F = 1 & I4: \{In = 1, Out = 1\} \Rightarrow F = 0
 \end{array}$$

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 in using the fuzzy logic environment FLOPER developed in our research group [MMPV10, MMPV11, MMPV12, JMM<sup>+</sup>13] (<http://dectau.uclm.es/floper/>). As we will explain in the rest of the paper, when FLOPER executes the following goal (representing formula  $F$ ) “ $(@not(i(In)) \& i(Out)) \mid (i(In) \& @not(i(Out)))$ ” with respect to a fuzzy

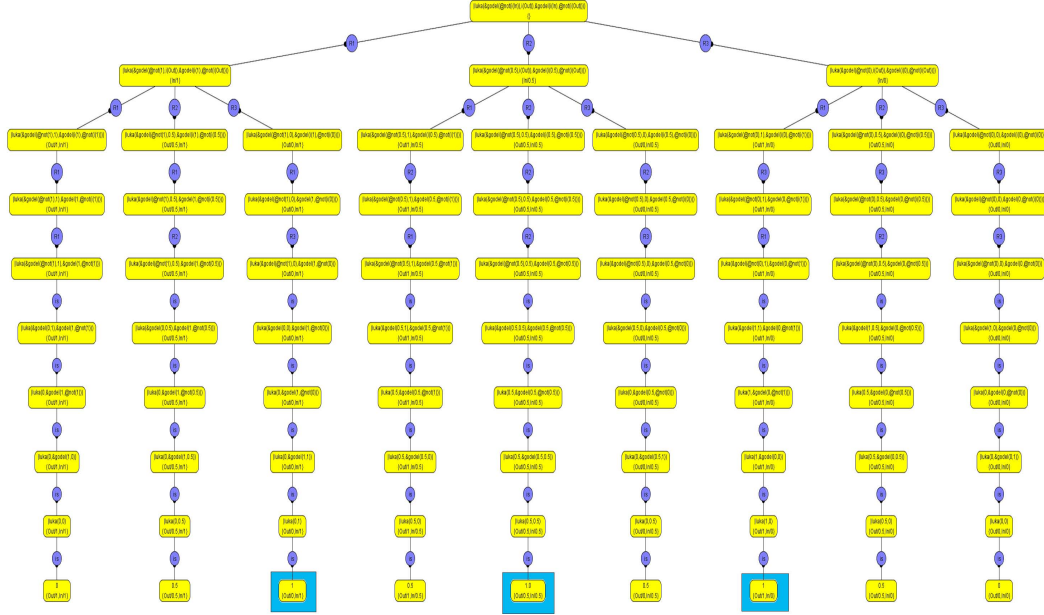


Figure 2: Interpreting a formula with three values for signals/propositions *In* and *Out*.

logic program composed by just two rules: “*i* (1) with 1” and “*i* (0) with 0”, it draws the tree shown in Figure 1, where models *I2* and *I3* appear in the two central leaves of the tree inside a blue box. Each branch in the tree starts by interpreting variables *In* and *Out* and continues with the evaluation of operators appearing in *F*.

Advice 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 3 afterwards):

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

Here we could use an SMT solver to prove that *F* is satisfiable. Following the approach of the work in [ABMV12], it can be easily checked that *F* is satisfiable<sup>1</sup> with the following SMT-LIB script, encoding the previous connectives into SAT modulo linear real arithmetic:

<sup>1</sup> The formula has infinite models of the form  $\{In = x, Out = y\}$  such that  $x + y = 1$ .

```

; Set logic: Quantifier Free Linear Real Arithmetic
(set-logic QF_LRA)

; min(x,y)
(define-fun min ((x Real) (y Real)) Real
  (ite (> x y) y x))

; x' = 1 - x
(define-fun agr_not ((x Real)) Real
  (- 1 x))

; &G(x,y) = min{x,y}
(define-fun and_godel ((x Real) (y Real)) Real
  (min x y))

; |L(x,y) = min{x+y,1}
(define-fun or_luka ((x Real) (y Real)) Real
  (min (+ x y) 1))

; Declaration of variables
(declare-fun x () Real)
(declare-fun y () Real)

; Ordering relation
(assert (>= x 0))
(assert (<= x 1))
(assert (>= y 0))
(assert (<= y 1))

; Formula to check
(assert (= (or_luka (and_godel (agr_not x) y)
                     (and_godel x (agr_not y)))
          1))

; Check for satisfiability
(check-sat)

```

It is easy to understand the SMT-LIB syntax of the previous code. Just in case, we recall that the `ite` expression corresponds to the *if-then-else* construct. Note that all necessary connectives are easily encoded as is done in [ABMV12]. It is worth noting that, apart from proving satisfiability of a formula, SMT solvers can be used to prove that a formula is a theorem, by checking unsatisfiability of its negation.

On the other hand, Figure 2 shows too three models in the tree depicted by FLOPER when considering only three kinds of values (that is, 0, 0.5 and 1) for interpreting *In* and *Out*. Such models include, apart of *I2* and *I3* seen before, the interpretation  $\{In = 0.5, Out = 0.5\}$  since, as

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

Figure 3: Conjunctors, disjunctors and implications from *Product*, *Gödel* and *Lukasiewicz* logics.

we can see in the detailed computations performed along the middle branch of the tree, we have:

$$\begin{array}{lll}
(0.5' \wedge 0.5) \vee (0.5 \wedge 0.5') & = & ((1 - 0.5) \wedge 0.5) \vee (0.5 \wedge 0.5') = (0.5 \wedge 0.5) \vee (0.5 \wedge 0.5') \\
= \min(0.5, 0.5) \vee (0.5 \wedge 0.5') & = & 0.5 \vee (0.5 \wedge 0.5') = 0.5 \vee (0.5 \wedge (1 - 0.5)) \\
= 0.5 \vee (0.5 \wedge 0.5) & = & 0.5 \vee \min(0.5, 0.5) = 0.5 \vee 0.5 \\
= \min(0.5 + 0.5, 1) & = & \min(1, 1) = 1
\end{array}$$

Similarly, we can check for instance in the second branch of the tree that  $\{In = 1, Out = 0.5\}$  is not a model (in fact, our chip can not return a signal of value 0.5 when its input is 1) since:

$$\begin{array}{lll}
(1' \wedge 0.5) \vee (1 \wedge 0.5') & = & ((1 - 1) \wedge 0.5) \vee (1 \wedge 0.5') = (0 \wedge 0.5) \vee (1 \wedge 0.5') = \\
\min(0, 0.5) \vee (1 \wedge 0.5') & = & 0 \vee (1 \wedge 0.5') = 0 \vee (1 \wedge (1 - 0.5)) = \\
0 \vee (1 \wedge 0.5) & = & 0 \vee \min(1, 0.5) = 0 \vee 0.5 = \\
\min(0 + 0.5, 1) & = & \min(0.5, 1) = 0.5
\end{array}$$

## 2 MALP, FLOPER and Automatic Theorem Proving

In what follows we describe a very simple *subset of the MALP<sup>2</sup> language* (see [MOV04, JMP09] for a complete formulation of this framework), which in essence consists in 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  $\leftarrow_1, \leftarrow_2, \dots$ ); conjunctive connectives ( $\wedge_1, \wedge_2, \dots$ ), disjunctive connectives ( $\vee_1, \vee_2, \dots$ ), and hybrid operators (usually denoted by  $@_1, @_2, \dots$ ), 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  $@(x_1, \dots, x_n)$  instead of  $@(x_1, \dots, @_{(x_{n-1}, x_n)})$ . By definition, the truth function for an n-ary aggregation operator  $[[@]] : L^n \rightarrow L$  is required to be monotonous.

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  $@(x_1, \dots, x_n) \triangleq E$ , where  $E$  is a  $L$ -expression not containing variable symbols apart from  $x_1, \dots, x_n$ . For instance, some fuzzy connective definitions in the lattice  $([0, 1], \leq)$  are presented in Figure 3 (from now on, this lattice

<sup>2</sup> Multi-Adjoint Logic Programming.

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  $B_1, \dots, B_n$  ( $n \geq 0$ ), 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 4), 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 on 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  $\sigma$  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:

$$\langle \mathcal{Q}[A]; \sigma \rangle \rightarrow_{AS} \langle (\mathcal{Q}[A/v])\theta; \sigma\theta \rangle$$

where  $A$  is the selected atom in  $\mathcal{Q}$ ,  $\theta = \text{mgu}(\{H = A\})^3$  and “ $H$  with  $v$ ” in  $\mathcal{P}$ . An *admissible derivation* is a sequence  $\langle \mathcal{Q}; id \rangle \rightarrow_{AS} \dots \rightarrow_{AS} \langle \mathcal{Q}'; \theta \rangle$ .

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** (Interpretive Step and Fuzzy Computed Answer) Let  $\mathcal{P}$  be a program,  $\mathcal{Q}$  a goal and  $\sigma$  a substitution. Assume that  $\llbracket @ \rrbracket$  is the truth function of connective  $@$  in the lattice  $(L, \leq)$  associated to  $\mathcal{P}$ , such that, for values  $r_1, \dots, r_n, r_{n+1} \in L$ , we have that  $\llbracket @ \rrbracket(r_1, \dots, r_n) = r_{n+1}$ . 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:

$$\langle \mathcal{Q}[\llbracket @ \rrbracket(r_1, \dots, r_n)]; \sigma \rangle \rightarrow_{IS} \langle \mathcal{Q}[\llbracket @ \rrbracket(r_1, \dots, r_n)/r_{n+1}]; \sigma \rangle$$

<sup>3</sup> Here  $\text{mgu}(E)$  denotes the *most general unifier* of an equation set  $E$  [LMM88].

An *interpretive derivation* is a sequence  $\langle \mathcal{Q}; \sigma \rangle \rightarrow_{IS} \dots \rightarrow_{IS} \langle \mathcal{Q}'; \sigma \rangle$ . When  $\mathcal{Q}' = r \in L$ , the state  $\langle r; \sigma \rangle$  is called a *fuzzy computed answer* (f.c.a.) for that derivation.

## 2.1 The Fuzzy Logic Programming Environment FLOPER

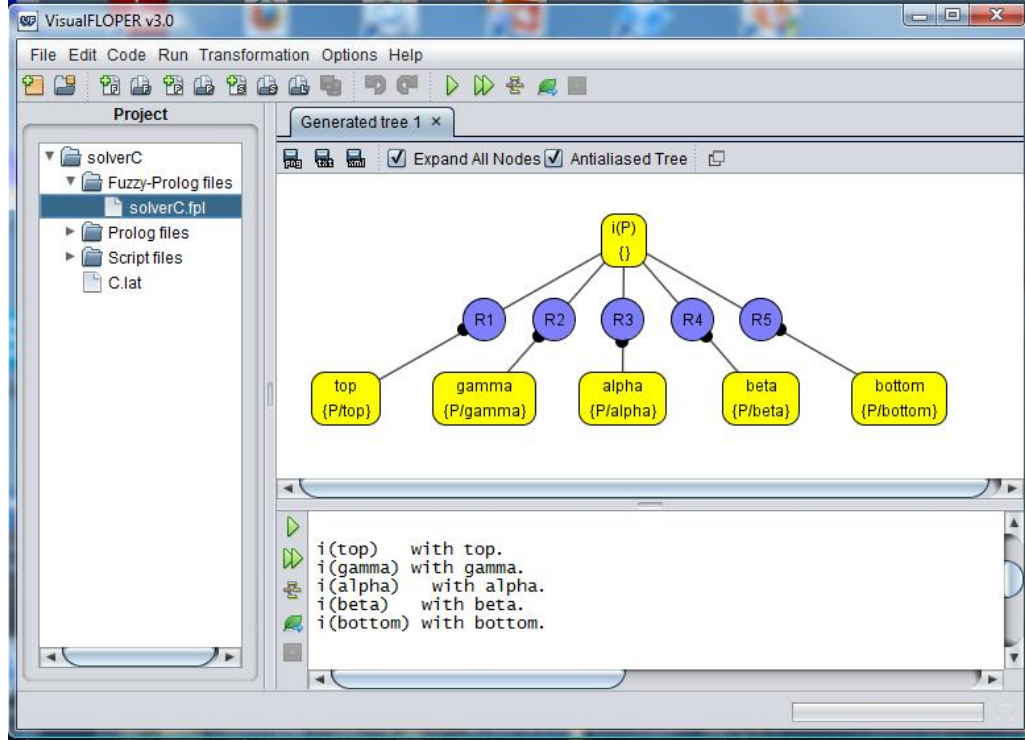


Figure 4: A work-session with FLOPER solving goal  $i(P)$ .

The parser of our FLOPER tool [MMPV10, MMPV11] 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, in [MMPV10] we explain that 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 also described in [MMPV10], according 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  $n_3$  re-

```

member(X) :- number(X), 0=<X, X=<1.                members([0,0.5,1]).
leq(X,Y)  :- X=<Y.                                bot(0).                    top(1).

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).
agr_not(X,Y)    :- pri_sub(1,X,Y).

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 5: Prolog code for representing lattice  $\mathcal{V}$ , which models truth degrees in the real interval  $[0, 1]$  with standard fuzzy connectives.

spectively, we must provide clauses defining the *connective predicates* “*and\_label<sub>1</sub>* / (*n<sub>1</sub>*+1)”, “*or\_label<sub>2</sub>* / (*n<sub>2</sub>*+1)” and “*agr\_label<sub>3</sub>* / (*n<sub>3</sub>*+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)*.

Following the Prolog style regulated by the previous guidelines, in Figure 5, we show the set of clauses modeling the more flexible lattice  $\mathcal{V}$ , which enables the possibility of working with truth degrees in the infinite space of the real numbers between 0 and 1, offering too a wide range of conjunction and disjunction operators recasted from the three typical fuzzy logics described before (i.e., the *Lukasiewicz*, *Gödel* and *product* logics), as well as a useful description for the hybrid aggregator *average* and the negation connective. Note that we have included the fact “*members([0,0.5,1])*.” which implies that in our application, propositional variables involved in fuzzy formulae to be proved, only three values (i.e., real numbers 0, 0.5 and 1, collected from the infinite unit interval) should be considered for interpreting such formulas, as shown in Figure 2.

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

|          |          |                                               |                               |
|----------|----------|-----------------------------------------------|-------------------------------|
|          |          | member(bottom) .                              | member(alpha) .               |
|          | T        | member(beta) .                                | member(gamma) .               |
|          |          | member(top) .                                 |                               |
|          |          | members([bottom,alpha,beta,gamma,top]) .      |                               |
|          |          | leq(bottom,X) .                               | leq(X,X) . leq(alpha,gamma) . |
|          | $\gamma$ | leq(beta,gamma) .                             | leq(X,top) . l(X,X) :- !.     |
|          |          | l(X,Y):-leq(X,Y),!. l(X,Y):-leq(X,Z),l(Z,Y) . |                               |
|          | / \      | and_godel(X,Y,Z) :- pri_inf(X,Y,Z) .          |                               |
|          |          | or_godel(X,Y,Z) :- pri_sup(X,Y,Z) .           |                               |
| $\alpha$ | $\beta$  | agr_impl(X,Y,Z) :- (l(Y,X),!,Z=top ; Z=X) .   |                               |
|          |          | pri_inf(X,Y,X):- l(X,Y) , !.                  |                               |
|          | \ /      | pri_inf(X,Y,Y):- l(Y,X) , !.                  |                               |
|          |          | pri_inf(-,-,bottom) .                         |                               |
|          | $\perp$  | pri_sup(X,Y,X):- l(Y,X) , !.                  |                               |
|          |          | pri_sup(X,Y,Y):- l(X,Y) , !.                  |                               |
|          |          | pri_sup(-,-,gamma) .                          |                               |

Figure 6: The finite, partially ordered lattice  $\mathcal{F}$  modeled in Prolog.

the bodies of clauses defining predicates with higher levels of expressivity (this is the case for instance, of the three kinds of fuzzy connectives we are considering: conjunctions, disjunctions and aggregations).

Since till now we have considered two classical, fully ordered lattices (with a finite and infinite number of elements, 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 lattice  $\mathcal{F}$  in the diagram of Figure 6, which is equipped with conjunction, disjunction and implication connectives based on the *Gödel* logic described in Figure 3, but with the particularity that now, in the general case, the *Gödel*'s conjunction must be expressed as  $\&_G(x,y) \triangleq \inf(x,y)$ , 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 6 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:

- Note that since truth degrees  $\alpha$  and  $\beta$  are incomparable, then any call to goals of the form “?- leq(alpha,beta) .” or “?- leq(beta,alpha) .” will always fail.
- A goal of the form “?- pri\_inf(alpha,beta,X) .”, instead of failing, successfully produces the desired result “X=bottom”.
- 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*”).

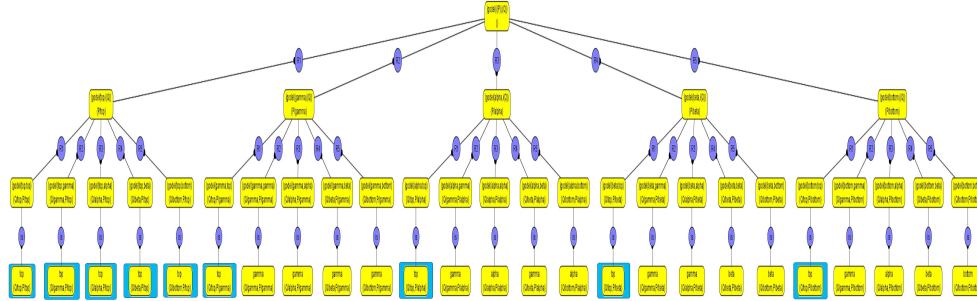
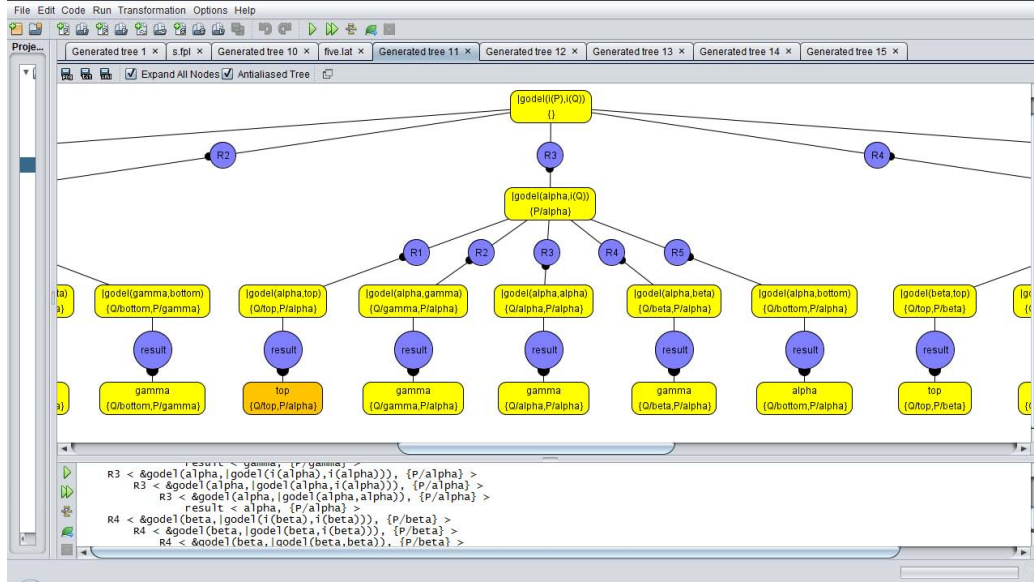


Figure 7: A work-session with FLOPER solving formula  $P \vee Q$  (25 interpretations, 9 models).

## 2.2 Some Examples

This subset of the MALP language 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  $\alpha$ ”, for each  $\alpha \in L$ . For instance, the MALP program associated to lattice  $\mathcal{F}$  in Figure 6 looks like:

```

i(top)           with      top.
i(gamma)         with      gamma.
i(alpha)         with      alpha.

```

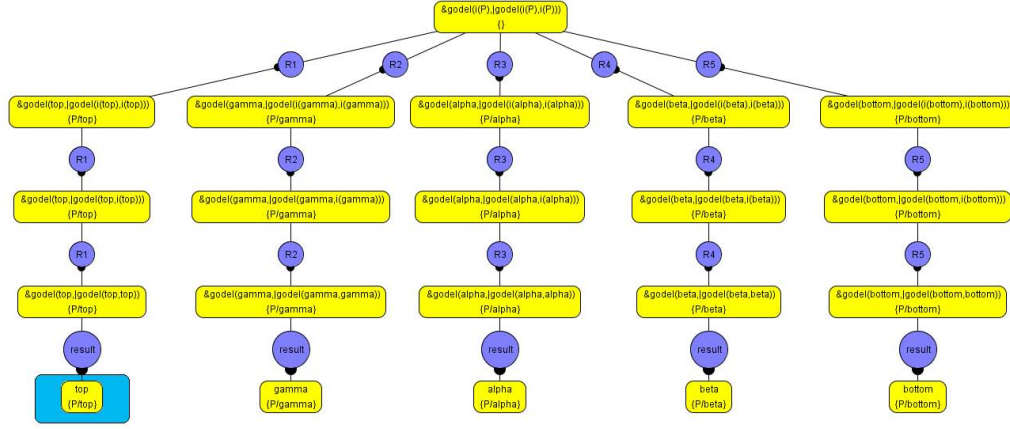


Figure 8: Full proof tree for formula  $P \wedge (P \vee P)$  with 1 model among 5 interpretations.

$i(\text{beta})$                       with         $\text{beta.}$   
 $i(\text{bottom})$                     with         $\text{bottom.}$

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 “ $\&\text{label}$ ” in goal  $F$ .
- For disjunctions, negations and implications, use respectively “ $|$ 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 6 and its residual MALP program just seen before. Firstly, if we execute goal “ $i(P)$ ” into FLOPER, we obtain the five interpretations for  $P$  shown in Figure 4. On the other hand, consider now the propositional formula  $P \vee Q$ , which is translated into the MALP goal “ $(i(P) \mid i(Q))$ ” and after being executed with FLOPER, the tool returns a tree<sup>4</sup> whose 25 leaves represent the whole set of interpretations (9 of them -inside blue clouds- are models) as seen in Figure 7. See also Figure 8 associated to formula  $P \wedge (P \vee P)$ .

<sup>4</sup> Each state contains its corresponding goal and substitution components and they are drawn inside yellow circles. Admissible steps, colored in blue, are labeled with the program rule they exploit. Finally, those blue circles annotated with word “ $is$ ”, correspond to interpretive steps. Sometimes we include blue circles labeled with “ $result$ ” which represents a chained sequence of interpretive steps.

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 the list of answers displayed in Figure 9, having all them the maximum truth degree “top”, which proves that this formula is a tautology, as wanted.

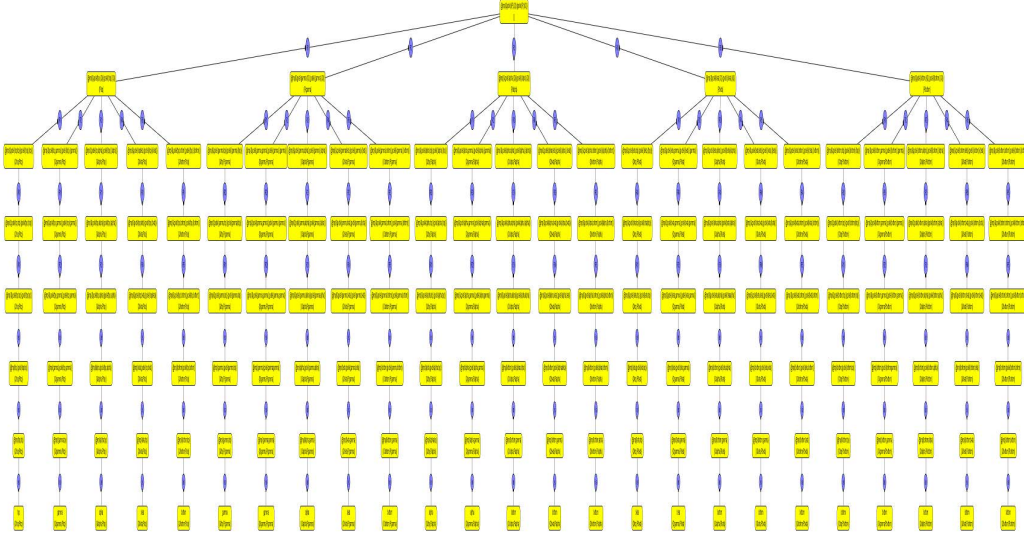


Figure 9: Full proof tree for tautology  $P \wedge Q \rightarrow P \vee Q$  (25 models).

### 2.3 Some Hints on Cost Measures

We wish to finish this section by providing some comments about cost measures and efficiency. So, given a lattice  $L$ , a formula  $F$  and its associated proof tree  $T$ , we define the following values:

- $v$  is the number of distinct variables in  $F$ .
- $v'$  is the number of occurrences (including repetitions) of variables in  $F$ .
- $c$  is the number of connectives in  $F$ .
- $r$  is the number of (marked) elements in lattice  $L$  given by predicate “members”.

And now we have that:

- The width of the tree  $T$ , or total number of interpretations of  $F$ , is  $r^v$ .
- The number of admissible steps performed on a single branch of  $T$  is  $v'$ .
- The number of interpretive steps performed on a single branch of  $T$  is  $c$ .

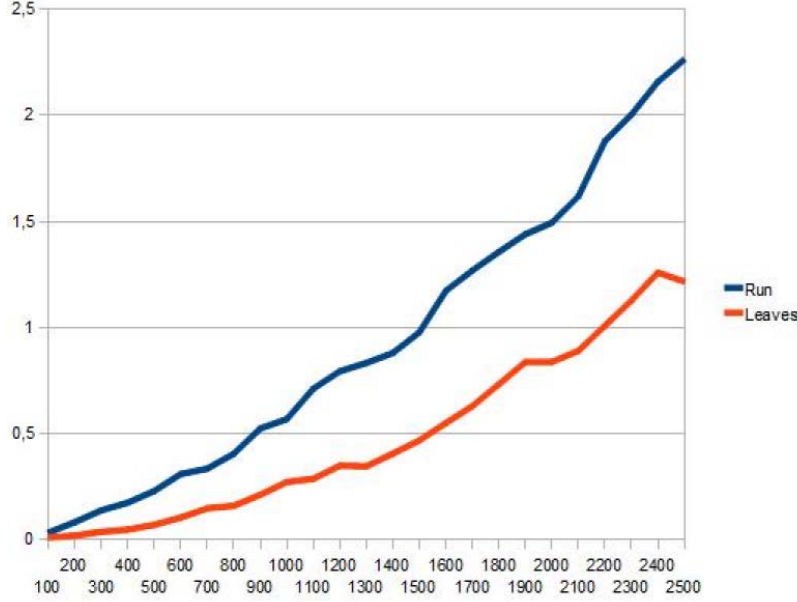


Figure 10: Behaviour of the method

- The depth of  $T$ , or number of computational (admissible/interpretive) steps for each possible interpretation of  $F$  is  $v' + c$ .
- An upper bound for the total number of admissible steps in  $T$  is  $|as| \leq (v' - v)r^v + \sum_{i=1}^v r^i$ .
- An upper bound for the total number of interpretive steps in  $T$  is  $|is| \leq cr^v$ .
- An finally, an upper bound for the total number of computational (admissible and interpretive) steps is  $|T| \leq (c + v' - v)r^v + \sum_{i=1}^v r^i$ .

Let us come back again to tautology  $P \wedge Q \rightarrow P \vee Q$  for which FLOPER displays the whole set of models seen in Figure 9, and assume now a more general version with the following shape  $P_1 \wedge \dots \wedge P_n \rightarrow P_1 \vee \dots \vee P_n$  for which we have studied its behaviour in the table of Figure 10. 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<sup>5</sup> needed to obtain the whole set of interpretations (all them are models in this case) for the formula. 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

<sup>5</sup> The benchmarks have been performed using a computer with processor Intel Core Duo, with 2 GB RAM and Windows Vista.

after compiling with FLOPER the MALP program and goal associated to our intended formula. The results achieved in the figure 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.

### 3 Conclusions and Future Work

In this paper we have focused on two different, but related problems regarding the automatic proving of propositional fuzzy formulae. In particular, whereas an SMT solver has been used for checking satisfiability, an alternative technique based on fuzzy logic programming has been introduced for finding the whole set of interpretations which are models for a given formula. In the future we plan to introduce improvements on both methods, regarding the set of truth degrees collected in the lattice used for interpreting a given formula. In the case of SMT, we plan to investigate how to cope with lattices equipped with partial ordering among their elements, whereas for the method based on fuzzy logic programming, it is important to design a much more flexible technique for dealing with infinite lattices than the one we have used in this paper (based on “pointing out” just a few number of truth degrees in the infinite space). In this last sense, some *halting rules* and *branch cuts* should be needed (maybe through *alfa cuts*) or even it could be interesting to study how to obtain all models of a formula by a constraint (as  $x + y = 1$  for the example given in the Introduction about the analogical chip) or a set of constraints.

**Acknowledgements:** Carlos Vázquez and Ginés Moreno received grants for International mobility from the University of Castilla-La Mancha (CYTEMA project and “Vicerrectorado de Profesorado”). Miquel Bofill and Mateu Villaret are partially supported by the *Ministerio de Economía y Competitividad* research project HeLo (ref. TIN2012-33042).

### Bibliography

- [ABMV12] C. Ansótegui, M. Bofill, F. Manyà, M. Villaret. 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.
- [Apt90] K. R. Apt. Logic Programming. In Leewen (ed.), *Handbook of Theoretical Computer Science*. Volume B: Formal Models and Semantics, chapter 10, pp. 493–574. MIT Press, Massachusetts Institute of Technology, USA, 1990.
- [Bra00] I. Bratko. *Prolog Programming for Artificial Intelligence*. Addison Wesley, September 2000.

- [BSST09] C. W. Barrett, R. Sebastiani, S. A. Seshia, C. Tinelli. Satisfiability Modulo Theories. In *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications 185, pp. 825–885. IOS Press, 2009.
- [Fit96] M. Fitting. *First-order logic and automated theorem proving*. Springer Verlag, 1996.
- [JMM<sup>+</sup>13] P. Julián, J. Medina, P. Morcillo, G. Moreno, 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*. Pp. 647–655. Springer Verlag, LNCS 7902, PART I, 2013.
- [JMP09] P. Julián, G. Moreno, J. Penabad. On the Declarative Semantics of Multi-Adjoint Logic Programs. In *Proceedings of the 10th International Work-Conference on Artificial Neural Networks, IWANN'09*. Pp. 253–260. Springer, Lecture Notes in Computer Science 5517, Salamanca, 2009.
- [Llo87] J. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second edition.
- [LMM88] J. L. Lassez, M. J. Maher, K. Marriott. Unification Revisited. In Minker (ed.), *Foundations of Deductive Databases and Logic Programming*. Pp. 587–625. Morgan Kaufmann, Los Altos, California, 1988.
- [MMPV10] P. Morcillo, G. Moreno, J. Penabad, C. Vázquez. A Practical Management of Fuzzy Truth Degrees using FLOPER. In al. (ed.), *Proceedings of 4th Intl Symposium on Rule Interchange and Applications, RuleML'10. Washington, USA, October 21–23*. Pp. 119–126. Springer Verlag, Lecture Notes in Computer Science 6403, 2010.
- [MMPV11] P. Morcillo, G. Moreno, J. Penabad, C. Vázquez. Fuzzy Computed Answers Collecting Proof Information. In al. (ed.), *Advances in Computational Intelligence. Proceedings of the 11th International Work-Conference on Artificial Neural Networks, IWANN'11*. Pp. 445–452. Springer Verlag, Lecture Notes in Computer Science 6692, 2011.
- [MMPV12] P. J. Morcillo, G. Moreno, J. Penabad, C. Vázquez. Dedekind-MacNeille completion and Cartesian product of multi-adjoint lattices. *Int. J. Comput. Math.* 89(13-14):1742–1752, 2012.
- [MOV04] J. Medina, M. Ojeda-Aciego, P. Vojtáš. Similarity-based Unification: a multi-adjoint approach. *Fuzzy Sets and Systems* 146:43–62, 2004.
- [Sti88] M. E. Stickel. A Prolog technology theorem prover: Implementation by an extended Prolog compiler. *Journal of Automated reasoning* 4(4):353–380, 1988.
- [VBG12] A. Vidal, F. Bou, L. Godo. 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 7520, pp. 633–640. 2012.