Using SAT/SMT Solvers for Efficiently Tuning Fuzzy Logic Programs

José A. Riaza and Ginés Moreno
Department of Computing Systems
University of Castilla-La Mancha, Albacete (02071), Spain
JoseAntonio.Riaza@uclm.es Gines.Moreno@uclm.es

***Abstract*—During the last years we have developed advanced tools for tuning fuzzy logic programs devoted to facilitate the selection of the more appropriate set of weights and fuzzy connectives used in programs rules. Designing accurate techniques for automating these tasks is very useful for programmers, even when they are time consuming. In order to increase its performance, in this paper we make use of powerful and well-known SAT/SMT solvers for improving our original approaches. Inspired by some previous experiences we have acquired in this setting, whose impact is growing in many modern software tools, we show some representative experiments (related to circuit validation and linear regression) and benchmarks which illustrate the significant advantages enjoyed by the new empowered method.**

***Index Terms*—Fuzzy Logic Programming, SAT/SMT, Tuning, Software for Soft Computing**

I. INTRODUCTION

Research on SAT (Boolean Satisfiability) and SMT (Satisfiability Modulo Theories) [9], [17] represents a successful and large tradition in the development of highly efficient automatic theorem provers for classic logic with a wide range of practical applications. There also exist attempts for covering fuzzy logics, as occurs with the approaches presented in [7], [30]. Moreover, if automatic theorem proving supposes both a starting point for the foundations of logic programming as well as one of its important application fields [16], [26], in [6], [10] we showed some preliminary guidelines about how fuzzy logic programming can face the automatic proving of fuzzy theorems by making use of the *''Fuzzy LOgic Programming Environment for Research"*, $\mathcal{FLOPER}$ in brief, developed in our research group.[1]. We have successfully used this tool for implementing real soft computing applications connecting with cloud computing [27]–[29], the semantic web [2]–[5] and, more recently, neural networks [21].

One step beyond, the main goal of the present paper is to make use of SMT solvers for reinforcing some tuning techniques we have recently implemented on this platform [20], [22]. The main reason for implementing our approach with MALP is the fact that, to the best of our knowledge, this is the only fuzzy logic programming language for which there exist tuning techniques already available.[2]

In this paper we focus on the so-called *multi-adjoint logic programming* approach, MALP in brief [14], [19], a powerful and promising approach in the area of fuzzy logic programming. In this framework, a program is a set of "weighted" rules whose bodies contain atoms connected by fuzzy connectives defined on a concrete lattice of truth degrees. Consider, for instance, the following MALP rule: "$good(X) \leftarrow_{\texttt{prod}} @_{\texttt{aver}}(nice(X), cheap(X))$ $with$ 0.8", where aggregator $@_{\texttt{aver}}$ is typically defined as $@_{\texttt{aver}}(x_1, x_2) \triangleq (x_1 + x_2)/2$. Therefore, the rule specifies that X is good—with a truth degree of 0.8— whenever X be nice and cheap enough. Assuming that X is nice and cheap with, e.g., truth degrees n and c, respectively, then X is good with a truth degree of $0.8 * ((n + c)/2)$.

To solve a MALP goal, i.e, a query to the system plus a substitution (initially the empty substitution, denoted by id), a generalization of the classical *modus ponens* inference rule called *admissible steps* ($\rightarrow_{AS}$), are systematically applied on atoms in a similar way to classical resolution steps in pure logic programming, thus returning a state composed by a computed substitution together with an expression where all atoms have been exploited. Next, this expression is interpreted under a given lattice by means of *interpretive steps* ($\rightarrow_{IS}$), hence returning a pair ⟨*truth degree*; *substitution*⟩, called *fuzzy computed answer* (fca), which is the fuzzy counterpart of the classical notion of computed answer used in pure logic programming (see [14], [19] for details).

When specifying a MALP program, it might sometimes be difficult to assign weights—truth degrees—to program rules, as well as to determine the right connectives. In order to overcome this drawback, in [20] we have recently introduced a symbolic extension of MALP programs called *symbolic multi-adjoint logic programming*, sMALP in brief. Here, we can write rules containing *symbolic* weights and *symbolic* connectives, i.e., truth degrees and operators –denoted as "$\#label$"– which are not defined on its associated multi-adjoint lattice. In order to evaluate these programs, we introduce a symbolic operational semantics that delays the evaluation of symbolic expressions. Therefore, a *symbolic fuzzy computed answer* –called sfca– could now include symbolic (unknown) truth values and connectives.

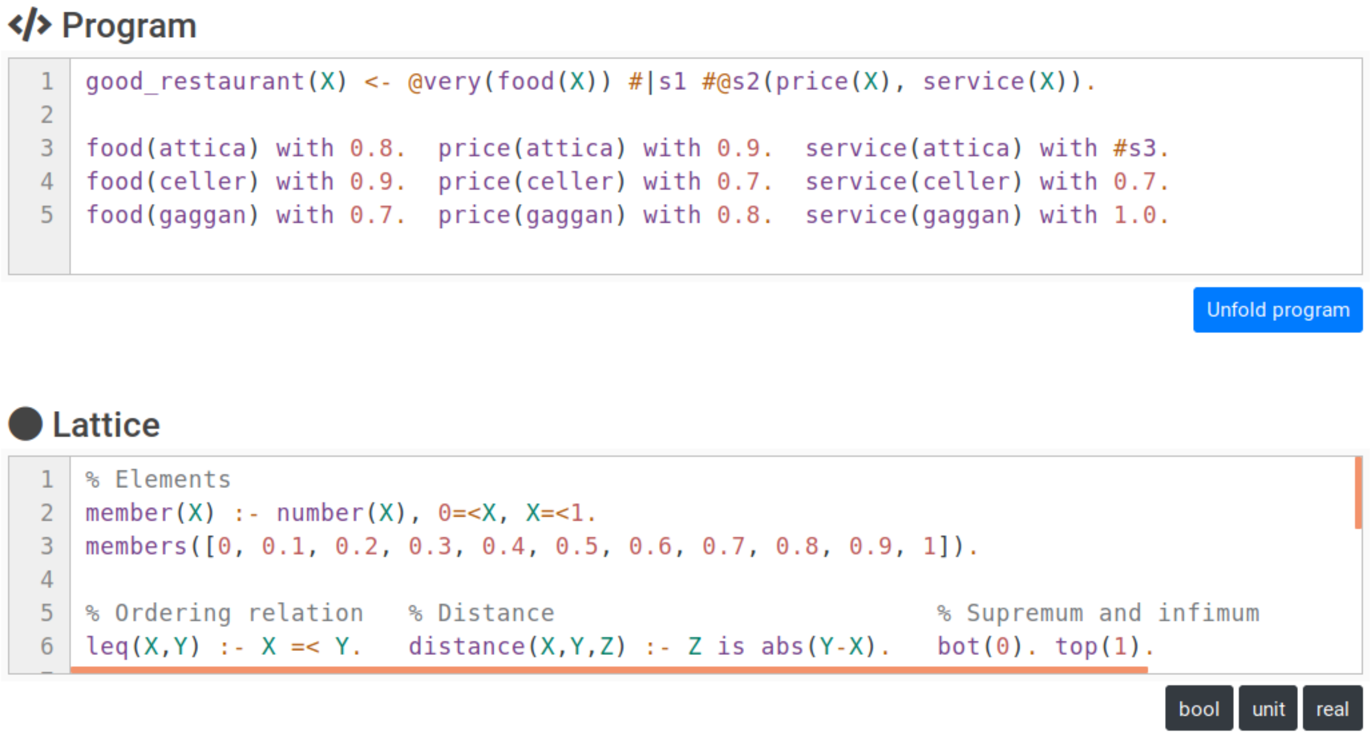
The approach is correct in the sense that using the symbolic semantics and then replacing the unknown values and connectives by concrete ones gives the same result as replacing these

---

[1] The online version of the system is available at `http://dectau.uclm.es/malp/sandbox`

[2] See [1] –as well as [25]– for analyzing a wide list of modern fuzzy software systems including the $\mathcal{FLOPER}$ system.

978-1-7281-6932-3/20/$31.00 ©2020 IEEE



```
good_restaurant(X) <- @very(food(X)) #|s1 #@s2(price(X), service(X)).

food(attica) with 0.8.  price(attica) with 0.9.  service(attica) with #s3.
food(celler) with 0.9.  price(celler) with 0.7.  service(celler) with 0.7.
food(gaggan) with 0.7.  price(gaggan) with 0.8.  service(gaggan) with 1.0.
```

```
% Elements
member(X) :- number(X), 0=<X, X=<1.
members([0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]).

% Ordering relation   % Distance                          % Supremum and infimum
leq(X,Y) :- X =< Y.   distance(X,Y,Z) :- Z is abs(Y-X).   bot(0). top(1).
```

Figure 1. Screenshot of the online tool available at `http://dectau.uclm.es/malp/sandbox`.

values and connectives in the original sMALP program and, then, applying the concrete semantics on the resulting MALP program. sMALP programs can be used to tune a program w.r.t. a given set of test cases, thus easing what is considered the most difficult part of the process: the specification of the right weights and connectives for each rule.

The SAT/SMT-based tuning technique we introduce in this paper makes use of one of the most popular solvers nowadays, i.e. Z3 [8], [23], according to the following three main tasks:

1) Firstly, the lattice of truth degrees of the sMALP program to be tuned is translated to Z3 syntax.
2) Next, the set of sfca's obtained after partially evaluating the test cases introduced a priori by the user are also automatically coded as a Z3 formula.
3) Finally, the Z3 solver is launched in order to minimize the deviation of the solutions w.r.t. the set of test cases while checking the satisfiability of such formula.

The structure of this paper is as follows. In Section II we recast from [20], [22] the original tuning method we have recently introduced in the $\mathcal{FLOPER}$ system. While Section III explains the new SAT/SMT-based reinforcement applied on such technique, Section IV illustrates the benefits of using the empowered tuning process focusing on two particular domains (circuit validation and linear regression). Finally, in Section V we show our conclusions and provide some lines for future research.

II. Tuning Symbolic Fuzzy Logic Programs

Let's now summarize the automated technique for tuning multi-adjoint logic programs using sMALP programs that we initially presented in [20] and next implemented in our online tool in [22]. We firstly introduce the running example of this section.

Example 1: At the bottom of Figure 1, we specify the lattice $([0,1], \leq)$ loaded by default in our freely accessible tool. In general, lattices are described by means of a set of Prolog clauses where the definition of the following predicates is mandatory: `member/1` and `members/1`, that identify the elements of the lattice; `bot/1` and `top/1` stand for the infimum and supremum elements of the lattice; and finally `leq/2`, that implements the ordering relation. Connectives are defined as predicates whose meaning is given by a number of clauses. The name of a predicate has the form `and_`*label*, `or_`*label* or `agr_`*label* depending on whether it implements a conjunction, a disjunction or an aggregator, where *label* is an identifier of that particular connective. The arity of the predicate is $n+1$, where n is the arity of the connective that it implements, so its last parameter is a variable to be unified with the truth value resulting of its evaluation.

Moreover, at the top of Figure 1, we can see a sMALP program loaded in the system. Here, we consider a travel guide that offers information about three restaurants, named *attica*, *celler* and *gaggan*, where each one of them is featured by three factors: the restaurant services, the quality of its food, and the price, denoted by predicates *service*, *food* and *price*, respectively. We assume that all weights can be easily obtained except for the weight of the fact $service(attica)$, which is unknown, as expressed by the symbolic weight $\#s3$. Since the programmer has also some doubts on the connectives to be used in the first rule, she introduces the symbolic disjunction and aggregator symbols $\#|s1$ and $\#@s2$.

Typically, a programmer has a model in mind where some parameters have a clear value. For instance, the truth value of a rule might be statistically determined and, thus, its value is easy to obtain. In other cases, though, the most appropriate values and/or connectives depend on subjective notions and, thus, programmers do not know how to obtain these values. In a typical scenario, we have an extensive set of *expected* computed answers (i.e., *test cases*), so the programmer can follow a "try and test" strategy. Unfortunately, this is a tedious and time consuming operation. Actually, it might even be impractical when the program should correctly model a large number of test cases.

The first action for initializing the tuning process consists in introducing a set of test cases with syntax: $r \to \mathcal{Q}$, where r is the desired truth degree for the fca associated to query $\mathcal{Q}$ (which obviously does not contain symbolic constants). For instance, in our running example we can introduce the following three test cases: $0.75 \to \mathtt{good\_restaurant}(\mathtt{attica})$, $0.8 \to \mathtt{good\_restaurant}(\mathtt{celler})$ and $0.9 \to \mathtt{good\_restaurant}(\mathtt{gaggan})$. Then, users simply need to click on the Tune program button for proceeding with the tuning process. The precision of the technique depends on the set of symbolic substitutions considered at tuning time. So, for assigning values to the symbolic constants, our tool takes into account all the truth values defined on a $\mathtt{members}/1$ predicate (which in our case is declared as $\mathtt{members}([0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]))$ as well as the set of connectives defined in the lattice associated to the program, which in our running example coincides with the three conjunction and disjunction connectives based on the so-called *Product*, *Gödel* and *Łukasiewicz* logics. Obviously, the larger the domain of values and connectives is, the more precise the results are. For tuning an sMALP program, we have implemented three methods, which exhibit different run-times (but they obviously produce the same outputs):

- Basic: The basic method is based on applying each symbolic substitution to the original sMALP program and then fully executing the resulting instantiated MALP programs (both the admissible and the interpretive stages).
- Symbolic: In this version, symbolic substitutions are directly applied to sfca's (thus, only the interpretive stage is repeatedly executed).
- Thresholded: In this case, we consider the symbolic algorithm improved with thresholding techniques.

The following definition formalizes the algorithms followed by the first couple of the tuning methods just commented, since the third one uses thresholding techniques for prematurely disregarding computations leading to non significant solutions and such improvements can be applied to both the basic and symbolic tuning methods.

```
0.85 -> good_restaurant(attica).
0.90 -> good_restaurant(celler).
0.95 -> good_restaurant(gaggan).
```

```
{#@s2/@aver, #|s1/|prod, #s3/0.3}
deviation: 0.050000000000000155
execution time: 49 milliseconds
```

Figure 2. Screenshot of the online tool after completing a tuning process.

Algorithms for tuning sMALP programs

Input: an sMALP program $\mathcal{P}$ and a number of (expected) test cases $(Q_i, \langle v_i; \theta_i \rangle)$[3], where Q_i is a goal and $\langle v_i; \theta_i \rangle$ is its expected fca for $i = 1, \ldots, k$.

Output: symbolic substitution Θ.

Basic method:

1) Consider a finite number of possible symbolic substitutions for $\mathsf{sym}(\mathcal{P})$, say $\Theta_1, \ldots, \Theta_n$, $n > 0$.
2) For each $j \in \{1, \ldots, n\}$, compute $\langle Q_i, \theta_i \rangle \to^* \langle v_{i,j}; \theta_i \rangle$ in $\mathcal{P}\Theta_j$, for $i = 1, \ldots, k$.
3) Return the symbolic substitution Θ_j that minimizes $\sum_{i=1}^{k} distance(v_i, v_{i,j})$.

Symbolic method:

1) For each test case $(Q_i, \langle v_i; \theta_i \rangle)$, compute the sfca $\langle Q'_i, \theta_i \rangle$ of $\langle Q_i, id \rangle$ in $\mathcal{P}$.
2) Consider a finite number of possible symbolic substitutions for $\mathsf{sym}(\mathcal{P})$, say $\Theta_1, \ldots, \Theta_n$, $n > 0$.
3) For each $j \in \{1, \ldots, n\}$, compute $\langle Q'_i\Theta_j, \theta_i \rangle \to_{IS}^{*} \langle v_{i,j}; \theta_i \rangle$, for $i = 1, \ldots, k$.
4) Return the symbolic substitution Θ_j that minimizes $\sum_{i=1}^{k} distance(v_i, v_{i,j})$.

As seen in Figure 2, the system also reports the processing time required by each method and offers an option for applying the best symbolic substitution to the original sMALP program in order to show the final, tuned MALP program.

---

[3] For readability, we usually write the simplified form $v_i \to Q_i$ by assuming that goal Q_i is a ground goal, that is, a goal without variables and thus, substitution θ_i is the identity -empty- substitution.

Example 2: In our case, the best substitution is $\Theta = \{\#|_{s1}/|_{\texttt{prod}}, \#@_{s2}/@_{\texttt{aver}}, \#_{s3}/0.3\}$ with a deviation of 0.05. By applying the symbolic substitution, we obtain a MALP program without symbolic constants, which can be executed w.r.t. a goal like `good_restaurant(X)`, thus obtaining the following three fca's: `<0.856, X/attica>`, `<0.943, X/celler>` and `<0.949, X/gaggan>`.

III. SAT/SMT FOR IMPROVING TUNING TECHNIQUES

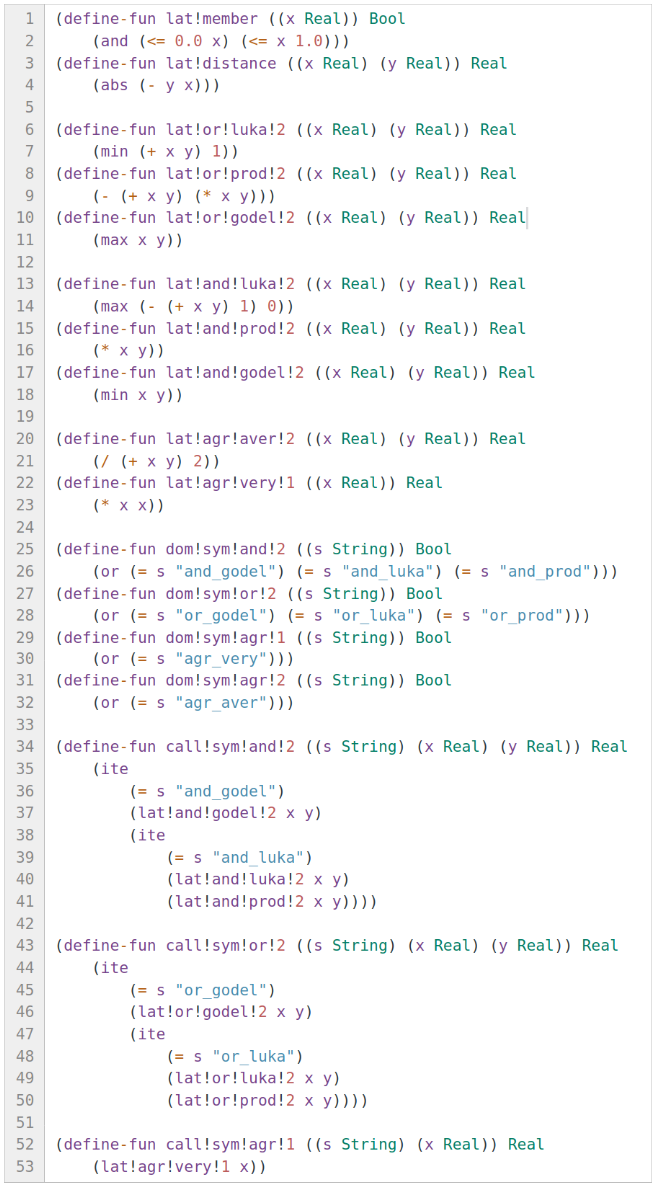
Boolean satisfiability (SAT) is the problem of checking if a propositional logic formula can ever evaluate to true [17]. Satisfiability modulo theories (SMT) generalizes boolean satisfiability by adding equality reasoning, arithmetic, quantifiers, and other useful first-order theories. An SMT solver is a tool for deciding the satisfiability of formulas in these theories. In this work, we have integrated the $\mathcal{FLOPER}$ system with Z3 [23], an SMT solver from Microsoft Research, with the aim of improving the performance of the tuning techniques previously implemented for manipulating MALP programs. Z3 supports the SMT-LIB [8] standard, which includes a scripting language that defines a textual interface for SMT solvers. In this section we describe how the MALP system translates a tuning problem into an SMT-LIB script that Z3 can solve in order to find the best symbolic substitution for a symbolic fuzzy logic program, given a set of test cases. Compared with the symbolic tuning algorithm seen in the previous section, the SMT-based method replaces steps 2), 3) and 4), that is, the new process starts after performing the first step which generates the set of sfca's $\langle Q'_i, \theta_i \rangle$ of $\langle Q_i, id \rangle$ in $\mathcal{P}$, being $(Q_i, \langle v_i; \theta_i \rangle)$ are the set of test cases.

As seen in Example 1, MALP associates to each fuzzy program a lattice of truth degrees. In order to perform the tuning process with Z3, it is necessary to translate the lattice associated with the program to SMT-LIB function declarations. Note that the predicate `members/1` is no longer necessary since Z3 will search for the most appropriate values from the underlying domain.

After partially executing the goal of each test case in the $\mathcal{FLOPER}$ environment, the resulting sMALP expressions not containing atoms are also translated to SMT-LIB, storing in a variable `deviation!` the sum of the distances between the sfca's and the expected truth degrees of the corresponding test cases, thus minimizing this variable in Z3, as follows:

```
(assert
  (= deviation!
    (+
      (lat!distance td_1 expr_1)
      (lat!distance td_2 expr_2)
      ...
      (lat!distance td_n expr_n))))
(minimize deviation!)
(check-sat)
(get-model)
```

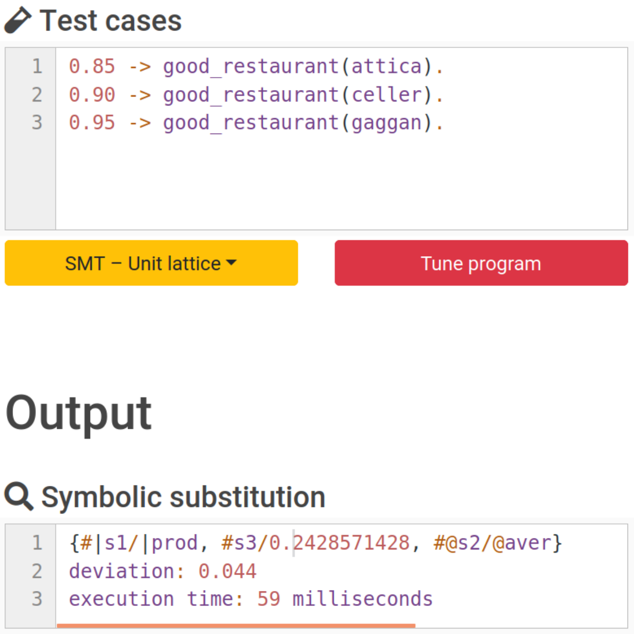
Example 3: The tuning process with Z3 shown in Figure 4 produces the following SMT-LIB script, where the symbolic connectives $\#|_{s1}$ and $\#@_{s2}$ are represented as strings `sym!or!2!s1` and `sym!agr!2!s2`, respectively, and the symbolic value $\#_{s3}$ is represented as a real number `sym!td!0!s3`:



```
(define-fun lat!member ((x Real)) Bool
    (and (<= 0.0 x) (<= x 1.0)))
(define-fun lat!distance ((x Real) (y Real)) Real
    (abs (- y x)))

(define-fun lat!or!luka!2 ((x Real) (y Real)) Real
    (min (+ x y) 1))
(define-fun lat!or!prod!2 ((x Real) (y Real)) Real
    (- (+ x y) (* x y)))
(define-fun lat!or!godel!2 ((x Real) (y Real)) Real
    (max x y))

(define-fun lat!and!luka!2 ((x Real) (y Real)) Real
    (max (- (+ x y) 1) 0))
(define-fun lat!and!prod!2 ((x Real) (y Real)) Real
    (* x y))
(define-fun lat!and!godel!2 ((x Real) (y Real)) Real
    (min x y))

(define-fun lat!agr!aver!2 ((x Real) (y Real)) Real
    (/ (+ x y) 2))
(define-fun lat!agr!very!1 ((x Real)) Real
    (* x x))

(define-fun dom!sym!and!2 ((s String)) Bool
    (or (= s "and_godel") (= s "and_luka") (= s "and_prod")))
(define-fun dom!sym!or!2 ((s String)) Bool
    (or (= s "or_godel") (= s "or_luka") (= s "or_prod")))
(define-fun dom!sym!agr!1 ((s String)) Bool
    (or (= s "agr_very")))
(define-fun dom!sym!agr!2 ((s String)) Bool
    (or (= s "agr_aver")))

(define-fun call!sym!and!2 ((s String) (x Real) (y Real)) Real
    (ite
        (= s "and_godel")
        (lat!and!godel!2 x y)
        (ite
            (= s "and_luka")
            (lat!and!luka!2 x y)
            (lat!and!prod!2 x y))))

(define-fun call!sym!or!2 ((s String) (x Real) (y Real)) Real
    (ite
        (= s "or_godel")
        (lat!or!godel!2 x y)
        (ite
            (= s "or_luka")
            (lat!or!luka!2 x y)
            (lat!or!prod!2 x y))))

(define-fun call!sym!agr!1 ((s String) (x Real)) Real
    (lat!agr!very!1 x))
```

Figure 3. Lattice $([0, 1], \leq)$ in SMT-LIB.

```
(declare-const sym!or!2!s1 String)
(declare-const sym!agr!2!s2 String)
(declare-const sym!td!0!s3 Real)
(declare-const deviation! Real)
(assert (dom!sym!or!2 sym!or!2!s1))
(assert (dom!sym!agr!2 sym!agr!2!s2))
(assert (lat!member sym!td!0!s3))
(assert (= deviation! (+
  (lat!distance 0.85
    (call!sym!or!2 sym!or!2!s1 0.65
      (call!sym!agr!2 sym!agr!2!s2 0.9 sym!td!0!s3)))
  (lat!distance 0.9
    (call!sym!or!2 sym!or!2!s1 0.81
      (call!sym!agr!2 sym!agr!2!s2 0.7 0.7)))
  (lat!distance 0.95
    (call!sym!or!2 sym!or!2!s1 0.49
      (call!sym!agr!2 sym!agr!2!s2 0.8 1))))))
```



Test cases

```
0.85 -> good_restaurant(attica).
0.90 -> good_restaurant(celler).
0.95 -> good_restaurant(gaggan).
```

Output

Symbolic substitution

```
{#|s1/|prod, #s3/0.2428571428, #@s2/@aver}
deviation: 0.044
execution time: 59 milliseconds
```

Figure 4. Screenshot of the online tool after completing a tuning process with Z3.

```
(minimize deviation!)
(check-sat)
(get-model)
```

The Z3 system produces the following output, whose model is interpreted by the $\mathcal{FLOPER}$ environment in order to generate the symbolic substitution $\Theta = \{\#|_{s_1}/|_{\text{prod}}, \#@_{s_2}/@_{\text{aver}}, \#_{s_3}/0.2428571428\}$ with a deviation of 0.044:

```
sat
(model
  (define-fun sym!or!2!s1 () String "or_prod")
  (define-fun sym!td!0!s3 () Real 0.2428571428?)
  (define-fun deviation! () Real 0.044)
  (define-fun sym!agr!2!s2 () String "agr_aver"))
```

IV. EXAMPLES

In this section we illustrate the use and benefits achieved on the afore mentioned SAT/SMT-based symbolic tuning technique by focusing on two well-known scenarios (circuit validation and linear regression), but it is important to indicate that the technique can be applied to any real world software application coded in MALP with the $\mathcal{FLOPER}$ environment.

A. Combinational equivalence checking

The problem of checking the equivalence of combinational circuits is an essential circuit design task. The simplest form of equivalence checking addresses combinational circuits. Let C_A and C_B denote two combinational circuits, both with inputs $x_1, \ldots, x_n$ and both with m outputs, C_A with outputs $y_1, \ldots, y_m$ and C_B with outputs $w_1, \ldots, w_m$. The function implemented by each one of the two circuits is defined as follows: $\mathbf{f_A} : \{0,1\}^n \to \{0,1\}^m$, and $\mathbf{f_B} : \{0,1\}^n \to \{0,1\}^m$. Let $\mathbf{x} \in \{0,1\}^n$ and define $\mathbf{f_A}(\mathbf{x}) = (f_{A,1}(\mathbf{x}), \ldots, f_{A,m}(\mathbf{x}))$ and $\mathbf{f_B}(\mathbf{x}) = (f_{B,1}(\mathbf{x}), \ldots, f_{B,m}(\mathbf{x}))$. The two circuits are not equivalent if the following condition holds:

$$\exists_{\mathbf{x} \in \{0,1\}^n} \; \exists_{1 \leq i \leq m} \; f_{A,i}(\mathbf{x}) \neq f_{B,i}(\mathbf{x})$$

which can be represented as the following satisfiability problem [18]:

$$\bigvee_{i=1}^{n} (f_{A,i}(\mathbf{x}) \oplus f_{B,i}(\mathbf{x})) = 1$$

The resulting satisfiability problem is illustrated in Figure 5, and it is referred to as a miter [11]. From these results, it is easy to encode in CNF the problem of verifying the equivalence of two combinational circuits and, therefore, it is capable of being tuned as a fuzzy logic program.

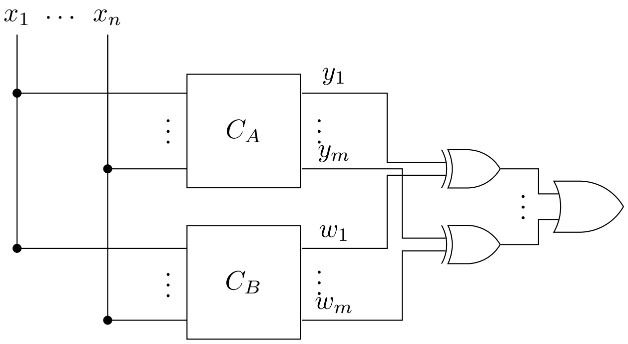


Figure 5. Equivalence Checking Miter

We will encode the combinational circuits in MALP as predicates of arity 2, where the first argument is a list containing the inputs, and the second one is a list containing the outputs. To check the equivalence between two circuits, we will implement the miter represented in Figure 5 as a predicate `miter/3` that takes two circuits (two atoms representing the name of the predicate of each circuit) and an input list, and checks if any of the outputs of both circuits is different for the input provided. `miter/3` is evaluated with a truth degree of `false` when all outputs are equal for the given input, or with a truth degree of `true` in any other case.

```
zip_xor([],[],[]).
zip_xor([X|Xs],[Y|Ys],[@xor(X,Y)|Zs]) :-
  zip_xor(Xs,Ys,Zs).

fold_or([],false).
fold_or([X|Xs],'|'(X,Ys)) :-
  fold_or(Xs,Ys).

miter(Ca,Cb,Xs) :-
  call(Ca,Xs,Ys),
  call(Cb,Xs,Ws),
  zip_xor(Ys,Ws,XOR),
  fold_or(XOR,OR),
  OR.
```

In order to check the equivalence between two circuits C_A and C_B with n inputs using the tuning technique, the system only needs a test case with the following shape:

```
true -> miter(CA, CB, [#x1, ..., #xm]).
```

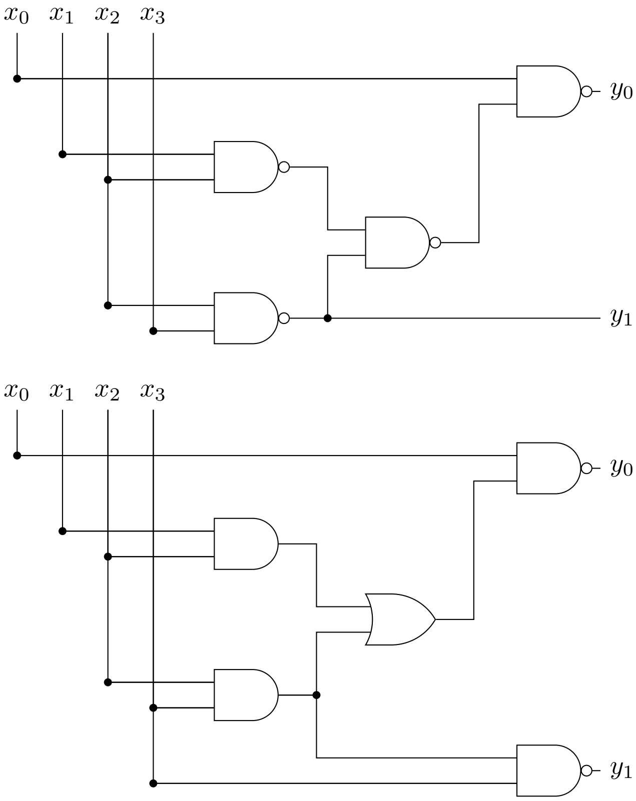


Figure 6. Two combinational circuits implementing the same function.

This test case tells the system that we want to find a combination of inputs $(x_1, \ldots, x_m)$ for C_A and C_B such that the output of the `miter/3` predicate is `true`, that is, such that some of the outputs y_i of both circuits are different. If the circuits are equivalent, the system will not be able to find such assignment of values, and it will return an arbitrary symbolic substitution with a deviation of 1.0. Otherwise, it will return a symbolic substitution for which one of the outputs is different in both circuits, with a deviation of 0.0.

Example 4: Let `a/2` and `b/2` be the MALP predicates representing the combinational circuits shown in Figure 6.

```
a([X0,X1,X2,X3], [Y0,Y1]) :-
  truth_degree(
    @not('&'(X0,
      @not('&'(@not('&'(X1,X2)), @not('&'(X2,X3))
    )))), Y0),
  truth_degree(@not('&'(X2,X3)), Y1).

b([X0,X1,X2,X3], [Y0,Y1]) :-
  truth_degree(
    @not('&'(X0,
      '|'(
        '&'(X1,X2), '&'(X2,X3)))), Y0),
  truth_degree(@not('&'('&'(X2,X3),X3)), Y1).
```

We run the tuning process with this program and with the following test case:

```
true -> miter(a, b, [#x0 , #x1 , #x2 , #x3]).
```

Table I
EXECUTION TIME (IN MILLISECONDS) OF THE TUNING ALGORITHMS IN MALP AND Z3 FOR CHECKING THE EQUIVALENCE OF COMBINATIONAL CIRCUITS BASED ON NUMBER OF INPUTS.

Inputs	MALP	Z3
4	45	36
5	930	37
6	2260	40
7	5670	41
8	12200	42
9	29340	43
10	65480	45

The MALP environment gives the following output, which demonstrates that the circuits are not equivalent, since the deviation of the best symbolic substitution found is 1.0:

```
{#x0/false, #x1/false, #x2/false, #x3/false}
deviation: 1.0
```

Example 5: Let us now introduce a third combinational circuit, not equivalent to the previous two ones, represented in MALP by the following predicate `c/2`:

```
c([X0,X1,X2,X3], [Y0,Y1]) :-
  truth_degree(@not('&'(X0 ,X1)), Y0),
  truth_degree(@not('|'(X2 ,X3)), Y1).
```

We run the tuning process to check the equivalence of the circuits `a/2` and `c/2`. In this case, the system finds a combination of the inputs, $(0, 0, 0, 1)$, for which both circuits produce different outputs, since the deviation is 0.0:

```
{#x0/false ,#x1/false ,#x2/false ,#x3/true}
deviation: 0.0
```

For this combination, circuit `a/2` produces the outputs $(1, 1)$, while circuit `c/2` produces the outputs $(1, 0)$.

Table I summarizes the averages of execution time in milliseconds[4] associated to the tuning algorithms when checking the equivalence of several combinational circuits by varying the number of inputs. It is worth mentioning that, apart from the fact that the tuning method based on Z3 is always better that the one not using it, as wanted, it also slightly increases its execution time when the number of inputs grow, while the MALP method (not using Z3) largely reduces its performance in an exponential way w.r.t. the number of input signals.

B. Linear regression

Linear regression is a linear approach for modelling the relationship between a dependent variable and one or more explanatory variables. Data consist of n observations on a dependent variable Y and p explanatory variables, $X_1, \ldots, X_p$. The relationship between Y an $X_1, \ldots, X_p$ is formulated as a linear model:

$$Y = \beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p + \varepsilon \tag{1}$$

---

[4] Each cell contains the average after 100 runs using a desktop computer equipped with an AMD Opteron™ processor @ 1593 MHz and 2.00 GB RAM.

Table II
Data set of Example 6.

Temp.	Noise	Temp.	Noise	Temp.	Noise
20.00	88.59	15.50	75.19	15.00	79.59
16.00	71.59	14.69	69.69	17.20	82.59
19.79	93.30	17.10	82.00	16.00	80.59
18.39	84.30	15.39	69.40	17.00	83.50
17.10	80.59	16.20	83.30	14.39	76.30

where $\beta_0, \beta_1, \ldots, \beta_p$ are constants referred to as the regression coefficients and ε is a random disturbance [12].

We will express the regression model as a MALP program with a single rule that defines Equation 1 by combining the connectives $\dot{|}_{add}(x, y) = x + y$ and $\dot{\&}_{prod}(x, y) = xy$ of the real lattice, where parameters β_i are symbolic constants:

$$\texttt{y(X1,...,Xp) <- \#b0 } |_{\texttt{add}} \texttt{ (\#b1 } \&_{\texttt{prod}} \texttt{ X1) } |_{\texttt{add}}$$
$$\texttt{... } |_{\texttt{add}} \texttt{ (\#bp } \&_{\texttt{prod}} \texttt{ Xp).}$$

This predicate takes p explanatory variables as inputs and is evaluated with a truth degree which is a linear combination of these inputs. To find the parameters β_i that best fit the data, we will tune this program by entering a test case for each sample in the data set, where the value of the dependent variable will be the expected truth degree of the test case.

Example 6: A popular, classical study described in [24], measured the frequency (the number of wing vibrations per second) of chirps made by a ground cricket, at various ground temperatures. The resulting data is shown in Table II, where the first column represents the temperature in Celsius scale, and the second one represents the noise in decibels. We want to analyse the relationship between the temperature and the noise level generated by the crickets. Since the data set has only one explanatory variable, i.e. temperature, the program will have two symbolic constants:

$$\texttt{chirps(Temp) <- \#b0 } |_{\texttt{add}} \texttt{ (\#b1 } \&_{\texttt{prod}} \texttt{ Temp).}$$

Each sample of the data set shown in Table II becomes a test case, thus obtaining the following set of test cases:

```
88.6 -> chirps(20.0). 71.6 -> chirps(16.0).
93.3 -> chirps(19.8). 84.3 -> chirps(18.4).
80.6 -> chirps(17.1). 75.2 -> chirps(15.5).
69.7 -> chirps(14.7). 82.0 -> chirps(17.1).
69.4 -> chirps(15.4). 83.3 -> chirps(16.2).
79.6 -> chirps(15.0). 82.6 -> chirps(17.2).
80.6 -> chirps(16.0). 83.5 -> chirps(17.0).
76.3 -> chirps(14.4).
```

After executing the tuning process, the MALP environment provides the following output:

```
{#b0/42.92, #b1/2.28}
deviation: 43.23
```

Figure 7 shows the linear regression model $y = 42.92 + 2.28x$ obtained by the MALP tuning technique using Z3.

Table III summarizes the averages of execution time[5] of the tuning algorithm with Z3, in seconds, when fitting linear regression models by varying the number of explanatory variables and the number of test cases.

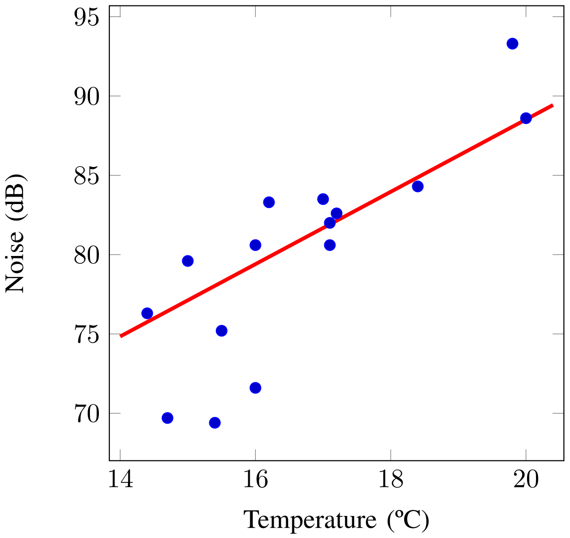


Figure 7. Linear regression model of Example 6.

Table III
Execution time (in seconds) of the tuning algorithm in Z3 for linear regression based on the number of explanatory variables and test cases.

		Explanatory variables				
		1	2	3	4	5
Test cases	20	0.35	1.55	6.27	11.38	33.48
	30	2.22	6.45	27.05	49.40	72.34
	40	6.01	25.32	107.68	165.52	378.15
	50	14.04	43.60	184.35	583.45	1547.56
	60	29.80	150.85	619.22	1094.20	3319.37
	70	40.30	213.32	794.55	2452.84	7532.63
	80	51.08	311.07	1058.35	4261.00	17317.25
	90	106.15	625.05	1658.60	7688.81	40313.90
	100	215.10	874.47	3189.72	11204.84	85873.49

V. Conclusions and Future Work

In this paper we have collected from [20], [22] our initial formulation of a symbolic extension of fuzzy logic programs belonging to the so-called *multi-adjoint logic programming* approach, as well as some tuning techniques useful for tailoring sMALP programs (also providing an online tool freely available via URL `http://dectau.uclm.es/malp/sandbox`). Next, inspired by our previous works [10] and [6], where we proposed two techniques for evaluating propositional fuzzy formulae with the $\mathcal{FLOPER}$ system in an alternative way than fuzzy SAT/SMT methods, we have focused on improving the former tuning techniques with the use of powerful SAT/SMT solvers.

Even when embedding the use of such solvers inside the core of our initial symbolic tuning algorithm has required some translation procedures for adapting the syntax of several MALP components (lattice of truth degrees, symbolic fuzzy computed answers, etc.) to the Z3 notation, the resulting performance of the new method has been highly improved.

---

[5] Each cell contains the average after 100 runs using a desktop computer equipped with an AMD Opteron™ processor @ 1593 MHz and 2.00 GB RAM.

Our benchmarks have revealed significant advantages in several aspects and, what is more important, although we have focused only on two well-known scenarios (circuit validation and linear regression) the technique is applicable to any other real world application coded in MALP with $\mathcal{FLOPER}$.

We are nowadays facing more involved non-linear regression problems [13]. Other pending task for the near future consists in exploring the synergies between our tuning approach and machine learning strategies and, since in [15] we have designed a new fuzzy language extending MALP with *similarity relations*, we also plan to enrich the present implementation of the SAT/SMT-based symbolic tuning technique to cope with $\mathcal{FASILL}$ programs managing similarities.

ACKNOWLEDGEMENTS

This work is partially supported by FEDER and the State Research Agency (AEI) of the MINECO Spanish Ministry under grant TIN2016-76843-C4-2-R (AEI/FEDER, UE).

REFERENCES

[1] J. Alcalá-Fdez and J. M. Alonso, "A Survey of Fuzzy Systems Software: Taxonomy, Current Research Trends, and Prospects," *IEEE Transactions on Fuzzy Systems*, vol. 24, no. 1, pp. 40–56, 2016.

[2] J. M. Almendros-Jiménez, A. Luna, and G. Moreno, "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

[3] ——, "Debugging while interpreting fuzzy xpath queries," in *Proc. of 2016 IEEE International Conference on Fuzzy Systems, FUZZ-IEEE'16, Vancouver, BC, Canada*. IEEE, 2016, pp. 233–240. [Online]. Available: https://doi.org/10.1109/FUZZ-IEEE.2016.7737692

[4] J. M. Almendros-Jiménez, A. Becerra-Terón, and G. Moreno, "Fuzzy queries of social networks with FSA-SPARQL," *Expert Syst. Appl.*, vol. 113, pp. 128–146, 2018. [Online]. Available: https://doi.org/10.1016/j.eswa.2018.06.051

[5] J. M. Almendros-Jiménez, A. Becerra-Terón, G. Moreno, and J. A. Riaza, "Tuning fuzzy SPARQL queries in a fuzzy logic programming environment," in *2019 IEEE International Conference on Fuzzy Systems, FUZZ-IEEE 2019, New Orleans, LA, USA*. IEEE, 2019, pp. 1–7. [Online]. Available: https://doi.org/10.1109/FUZZ-IEEE.2019.8858958

[6] J. M. Almendros-Jiménez, M. Bofill, A. L. Tedesqui, G. Moreno, C. Vázquez, and M. Villaret, "Fuzzy xpath for the automatic search of fuzzy formulae models," in *Scalable Uncertainty Management - Proc. of the 9th International Conference, SUM'15*, ser. Lecture Notes in Computer Science, vol. 9310. Springer, 2015, pp. 385–398. [Online]. Available: https://doi.org/10.1007/978-3-319-23540-0_26

[7] C. Ansótegui, M. Bofill, F. Manyà, and 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*, 2012, pp. 25–30.

[8] C. Barrett, A. Stump, and C. Tinelli, "The smt-lib standard: Version 2.0," in *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, vol. 13, 2010, p. 14.

[9] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability Modulo Theories," in *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009, vol. 185, pp. 825–885.

[10] M. Bofill, G. Moreno, C. Vázquez, and M. Villaret, "Automatic proving of fuzzy formulae with fuzzy logic programming and SMT," in *Proc. of XIII Conference on Programming and Languages, PROLE'13*. Extended version published by ECEASST, volume 64, pages: 1-19 (available at `http://dx.doi.org/10.14279/tuj.eceasst.64.991`), 2013, pp. 151–165.

[11] D. Brand, "Verification of large synthesized designs," in *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*. IEEE, Nov 1993, pp. 534–537.

[12] S. Chatterjee and A. S. Hadi, *Regression analysis by example*. John Wiley & Sons, 2015.

[13] M. J. Gacto, J. M. Soto-Hidalgo, J. Alcalá-Fdez, and R. Alcalá, "Experimental Study on 164 Algorithms Available in Software Tools for Solving Standard Non-Linear Regression Problems," *IEEE Access*, vol. 7, pp. 108 916–108 939, 2019.

[14] P. Julián-Iranzo, 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.

[15] ——, "Thresholded semantic framework for a fully integrated fuzzy logic language," *J. Log. Algebr. Meth. Program.*, vol. 93, pp. 42–67, 2017. [Online]. Available: https://doi.org/10.1016/j.jlamp.2017.08.002

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

[17] S. Malik and G. Weissenbacher, "Boolean satisfiability solvers: techniques and extensions," in *Software Safety and Security: Tools for Analysis and Verification*. IOS Press, 2012.

[18] J. Marques-Silva, "Practical applications of boolean satisfiability," in *2008 9th International Workshop on Discrete Event Systems*, May 2008, pp. 74–80.

[19] 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.

[20] G. Moreno, J. Penabad, J. Riaza, and G. Vidal, "Symbolic execution and thresholding for efficiently tuning fuzzy logic programs," in *Logic-Based Program Synthesis and Transformation - 26th International Symposium, LOPSTR 2016, Edinburgh, UK, September 6-8, 2016, Revised Selected Papers*, 2016, pp. 131–147. [Online]. Available: https://doi.org/10.1007/978-3-319-63139-4_8

[21] G. Moreno, J. Pérez, and J. A. Riaza, "Fuzzy logic programming for tuning neural networks," in *Rules and Reasoning - proc. of the Third International Joint Conference, RuleML+RR 2019*, ser. Lecture Notes in Computer Science, vol. 11784. Springer, 2019, pp. 190–197. [Online]. Available: https://doi.org/10.1007/978-3-030-31095-0_14

[22] G. Moreno and J. A. Riaza, "An online tool for tuning fuzzy logic programs," in *Rules and Reasoning - International Joint Conference, RuleML+RR 2017, London, UK, July 12-15, 2017, Proceedings*, ser. Lecture Notes in Computer Science, S. Costantini, E. Franconi, W. V. Woensel, R. Kontchakov, F. Sadri, and D. Roman, Eds., vol. 10364. Springer, 2017, pp. 184–198. [Online]. Available: https://doi.org/10.1007/978-3-319-61252-2_13

[23] L. D. Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[24] G. W. Pierce, *The Songs of Insects; with Related Material on the Production, Propagation, Detection, and Measurement of Sonic and Supersonic Vibrations*. Harvard University Press, 1946.

[25] J. M. Soto-Hidalgo, J. M. Alonso, G. Acampora, and J. Alcalá-Fdez, "JFML: A Java Library to Design Fuzzy Logic Systems According to the IEEE Std 1855-2016," *IEEE Access*, vol. 6, pp. 54 952–54 964, 2018.

[26] M. E. Stickel, "A prolog technology theorem prover: Implementation by an extended prolog compiler," *Journal of Automated reasoning*, vol. 4, no. 4, pp. 353–380, 1988.

[27] L. Tomás, C. Vázquez, J. Tordsson, and G. Moreno, "Reducing noisy-neighbor impact with a fuzzy affinity-aware scheduler," in *2015 International Conference on Cloud and Autonomic Computing, Boston, MA, USA, September 21-25, 2015*. IEEE Computer Society, 2015, pp. 33–44. [Online]. Available: https://doi.org/10.1109/ICCAC.2015.14

[28] C. Vázquez, G. Moreno, L. Tomás, and J. Tordsson, "A cloud scheduler assisted by a fuzzy affinity-aware engine," in *Proc. of 2015 IEEE Int. Conference on Fuzzy Systems, FUZZ-IEEE'15, Istanbul, Turkey*. IEEE, 2015, pp. 1–8. [Online]. Available: http://dx.doi.org/10.1109/FUZZ-IEEE.2015.7337931

[29] C. Vázquez, L. Tomás, G. Moreno, and J. Tordsson, "A fuzzy approach to cloud admission control for safe overbooking," in *Proc. of 10th Int. Workshop of Fuzzy Logic and Applications, WILF'13, Genoa, Italy*, ser. LNCS, vol. 8256. Springer, 2013, pp. 212–225. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-03200-9_22

[30] A. Vidal, F. Bou, and L. Godo, "An SMT-Based Solver for Continuous t-norm Based Logics," in *Proceedings of the 6th International Conference on Scalable Uncertainty Management*, ser. Lecture Notes in Computer Science, vol. 7520, 2012, pp. 633–640.