

The Implementation of a Product Fuzzy DPLL Solver

Ivor Uhliarik^a

Department of Applied Informatics, Comenius University, Mlynská dolina, 842 48 Bratislava, Slovakia

Keywords: Fuzzy Logic, Product Logic, Automated Theorem Proving, DPLL, Satisfiability Solving.

Abstract: The area of automated theorem proving in fuzzy logics has been revisited during the last decade with novel approaches, mathematical foundations, and software. However, only a few of these are capable of dealing with logics based on the product t-norm. The existing methods are usually based on (1) translations into satisfiability modulo theories, (2) evolutionary algorithms, and (3) fuzzy extensions of classical-logic procedures, such as the Davis-Putnam-Logemann-Loveland (DPLL) procedure. In this paper we present the results of our work on the first DPLL-based solver for product propositional logic extended with the Monteiro-Baaz Δ operator and order ($\prec, =$) operators. Our contribution consists of the refinement and completion of our previously proposed deterministic algorithm and the working implementation of the solver. Comparing to other approaches, the essential difference of ours lies in its self-containment—it is not based on translations into other systems, which provides possibilities for feasible modifications or further optimizations. The solver yields answers to the 1-satisfiability and validity problems, and is available for download and use as free and open-source software.

1 INTRODUCTION AND RELATED WORK


The three prominent fuzzy logics are based on the Gödel, Łukasiewicz, and product t-norms. To this day, product fuzzy logic remains the least explored in terms of automated theorem proving and applications. In spite of this, recent years have witnessed the emerging of approaches that deal with product logic not only on the propositional level (Guller, 2013; Brys et al., 2013), but also with modal extensions (Vidal, 2016), description logics (Bobillo and Straccia, 2007; Béjar et al., 2013), and in fuzzy answer set programming (Janssen et al., 2012; Alviano and Peñaloza, 2015).

In propositional logic, the field of automated theorem proving revolves around the task of proving the satisfiability (SAT) or validity (VAL) of a formula by a solver. One such solver (Vidal, 2016) is based on satisfiability modulo theories. In addition to product propositional logic, it is able to operate over any fuzzy logic with a continuous t-norm by the Mostert-Shields theorem (Mostert and Shields, 1957). The solver is mature in the sense that it has been thoroughly described, validated, evaluated, and its implementation is available to the public.

A different kind of proposal for SAT solving in product propositional logic (Guller, 2013) extends the Davis-Putnam-Logemann-Loveland (DPLL) (Davis et al., 1962) procedure known in Boolean logic to the fuzzy case. The approach establishes an analogy to the branching and simplifying rules in classical DPLL. In contrast to approaches relying on translations into other systems and their solvers, this procedure could be used to develop a self-contained, transparent solver—one with implementation flexible at opting to use various kinds of rules as heuristics, or even using machine learning techniques to guide the DPLL tree traversal (Suttner and Ertl, 1990).

In this paper we present the next milestone of our existing research. We choose to work with the product fuzzy DPLL procedure described above. In the previous iteration (Uhliarik, 2019) we proposed the initial version of the algorithm for performing an informed search of the fuzzy DPLL tree. Herewith we propose a refinement of the algorithm, making it simpler and more efficient, and complete the previous contribution by proposing a method to solve the unit contradiction problem. Also, we introduce the working implementation of our approach.

One of the aspects of product logic that makes it worthwhile to investigate is the embeddability of Łukasiewicz and (extended) Gödel logic within product logic extended by the operator Δ (Baaz et al.,

^a  <https://orcid.org/0000-0002-0495-5467>

1998). The idea to develop a solver around the product t-norm that would be able to work uniformly with the three logics and, by the Mostert-Shields theorem, also all other continuous t-norm-based ones, is indeed attractive. This is the reason we choose to work with Δ -extended product logic.

The rest of this paper is structured as follows. In Section 2 we revisit the preliminary notions that are necessary to understand later text. Section 3 describes the algorithms performing the translation of formulae into normal form and the informed search of the DPLL tree. In the same section we also present our full approach to applying the unit contradiction rule, one of the fuzzy DPLL rules. Next, in Section 4 we describe our implementation of the solver. Section 5 lays down the information resulting from the evaluation of the implementation. Finally, we conclude the paper in Section 6 and list future work.

2 PRELIMINARIES

In this section we introduce the concepts needed for the understanding of later sections of this paper. We define the syntax and semantics of Δ -extended product propositional logic, the order clausal form and its elements, and the product fuzzy DPLL procedure.

2.1 Product Propositional Logic

This work is focused on dealing with product propositional logic extended with the Monteiro-Baaz Δ operator, interpreted by the standard product algebra extended with the operators \equiv , \prec , and Δ :

$$\Pi_{\Delta} = ([0, 1], \leq, \vee, \wedge, \cdot, \Rightarrow, \neg, \equiv, \prec, \Delta, 0, 1)$$

The connectives of the logic are \neg (negation), \wedge (conjunction), $\&$ (strong conjunction), \vee (disjunction), \rightarrow (implication), \leftrightarrow (equivalence), \equiv (equality), \prec (strict order), and Δ (delta)¹, with the associated operations of \neg , \wedge , \cdot , \vee , \Rightarrow , \equiv , \prec , and Δ defined as

$$\begin{aligned} x \Rightarrow y &= \begin{cases} 1 & \text{if } x \leq y, \\ \frac{y}{x} & \text{else;} \end{cases} & \neg x &= \begin{cases} 1 & \text{if } x = 0, \\ 0 & \text{else;} \end{cases} \\ x \equiv y &= \begin{cases} 1 & \text{if } x = y, \\ 0 & \text{else;} \end{cases} & x \prec y &= \begin{cases} 1 & \text{if } x < y, \\ 0 & \text{else;} \end{cases} \\ \Delta x &= \begin{cases} 1 & \text{if } x = 1, \\ 0 & \text{else;} \end{cases} \end{aligned}$$

¹With the decreasing connective precedence: \neg , Δ , $\&$, \equiv , \prec , \wedge , \vee , \rightarrow , \leftrightarrow .

\vee , \wedge as the supremum and infimum operator on $[0, 1]$ respectively, and \cdot as the algebraic product. The equivalence connective in the expression $x \leftrightarrow y$ is interpreted as $x \Rightarrow y \wedge y \Rightarrow x$. 0 and 1 are the symbols representing the truth constants *false* and *true*, interpreted in the algebra as its absorbing and neutral element.

The residuum operator \Rightarrow satisfies the residuation principle w.r.t. operation \cdot ; for any $x \in \Pi_{\Delta}$, the negation \neg satisfies the condition $\neg x = x \Rightarrow 0$, and Δ satisfies the condition $\Delta x = x \equiv 1$.

To construct formulae we follow the convention of (Guller, 2013; Uhliarik, 2019; Guller, 2020): let *PropAtom* be the set of propositional atoms. Order propositional formulae are constructed from *PropAtom*, the truth constants 0 and 1, and the logical connectives. Next, let *OrdPropForm* be the set of all such formulae. Any subset of *OrdPropForm* is called an order theory.

As in (Guller, 2013), we assume the valuation of propositional atoms to be the mapping $\mathcal{V} : \text{PropAtom} \rightarrow [0, 1]$ such that $\mathcal{V}(0) = 0$ and $\mathcal{V}(1) = 1$. For any formula $\varphi \in \text{OrdPropForm}$, the value $\|\varphi\|^{\mathcal{V}} \in [0, 1]$ of φ in \mathcal{V} is defined recursively on the structure of φ by the respective application of the unary and binary operations corresponding to the connectives in φ . More formally, the following applies according to the structure of φ :

$$\begin{aligned} \varphi \in \text{PropAtom}, \|\varphi\|^{\mathcal{V}} &= \mathcal{V}(\varphi); \\ \varphi = \neg\varphi_1, \|\varphi\|^{\mathcal{V}} &= \neg\|\varphi_1\|^{\mathcal{V}}; \\ \varphi = \Delta\varphi_1, \|\varphi\|^{\mathcal{V}} &= \Delta\|\varphi_1\|^{\mathcal{V}}; \\ \varphi = \varphi_1 \diamond \varphi_2, \|\varphi\|^{\mathcal{V}} &= \|\varphi_1\|^{\mathcal{V}} \diamond \|\varphi_2\|^{\mathcal{V}}, \\ &\diamond \in \{\wedge, \&, \vee, \rightarrow, \equiv, \prec\}; \\ \varphi = \varphi_1 \leftrightarrow \varphi_2, \|\varphi\|^{\mathcal{V}} &= (\|\varphi_1\|^{\mathcal{V}} \Rightarrow \|\varphi_2\|^{\mathcal{V}}) \cdot \\ &(\|\varphi_2\|^{\mathcal{V}} \Rightarrow \|\varphi_1\|^{\mathcal{V}}). \end{aligned}$$

The subject of the next sections will be the problem of satisfiability and validity (tautologicity) of formulae. We can define these on a formal level using the concept of valuation. For the formula $\varphi \in \text{OrdPropForm}$, we say φ has the model \mathcal{V} (is true in \mathcal{V}), or $\mathcal{V} \models \varphi$ iff $\|\varphi\|^{\mathcal{V}} = 1$. A formula is satisfiable iff there exists some model of it, and is a tautology iff every valuation is its model.

The theory T has the model \mathcal{V} , or $\mathcal{V} \models T$ iff $\mathcal{V} \models \varphi$ for all $\varphi \in T$. A theory is satisfiable iff it has a model, and is valid iff every valuation is its model.

Finally, for two formulae $\varphi, \varphi' \in \text{OrdPropForm}$, φ is equivalent to φ' , or $\varphi \equiv \varphi'$ iff $\|\varphi\|^{\mathcal{V}} = \|\varphi'\|^{\mathcal{V}}$ for every valuation \mathcal{V} .

2.2 Order Clausal Form

Automated theorem provers in classical propositional logic are mostly known to require the input formulae to be in a canonical form, usually the conjunctive or disjunctive normal form. Such also applies to the product propositional solver. To facilitate uniform and efficient processing using the fuzzy extension of the DPLL procedure over Π_Δ , the input is expected to be in the *order clausal form*. As in the case of Boolean normal forms, there exists a method of translation of an arbitrary formula into this form, which will be described in Section 3. Here we enumerate the building blocks of the form, staying consistent with its proposal (Guller, 2013), and we will refer to them later in the text.

The set of propositional atoms $PropAtoms$ and the truth constants 0, 1 have already been defined.

The *power of atom* in the form a^n is the n -th power of atom a interpreted by the \cdot operator.

The *conjunction* C_n is the set of powers of atoms given in the form $a_1^{p_1} \& \dots \& a_n^{p_n}$, $n \geq 1$. The atoms a_i in the expression must be unique (cannot occur more than once). $PropConj$ is the set of all conjunctions.

The expression of the form $\varepsilon_1 \diamond \varepsilon_2$ where $\diamond \in \{=, \prec\}$ and $\varepsilon_i \in PropConj \cup \{0, 1\}$ is called the *order literal*. More specifically, the literal associated with the connective $=$ is the *equality literal* and \prec the *strict order literal*. A literal is *pure* iff it does not contain any of the constants 0, 1.

The set of literals $l_1 \vee \dots \vee l_n$ is the *order clause*. \square will denote the *empty clause*, and we refer to $\{l\}$ as the *unit clause*.

A set of clauses constructs the *order clausal theory*. Further, an order clausal theory is $\{\text{pure, unit}\}$ iff it contains only $\{\text{pure, unit}\}$ clauses.

A more comprehensive list of formal definitions of terms may be found in (Guller, 2013, Sect. 3), together with the interpolation rules used to perform the translation, the discussion of technical aspects, and the complexity of the algorithm.

2.3 Product DPLL Procedure

The original DPLL procedure (Davis et al., 1962) on classical propositional logic is a popular, well-established backtracking search algorithm that solves the satisfiability problem given a formula by attempting to find its model (or proving its unsatisfiability). The procedure may be thought of as a tree-traversing algorithm, where the tree is split into two branches according to the possible valuations of literals. Moreover, the method uses two rules at each step to con-

strain the search space: unit propagation and pure literal elimination.

For the sake of visualizing analogy, the branching step of DPLL may be written in the following form:

$$\frac{S}{S \cup \{l\} \mid S \cup \{-l\}} \quad (\text{Branching rule})$$

for literal l occurring in theory S .

The core idea of the fuzzy extension of the procedure introduced by (Guller, 2013) and refined in (Guller, 2020) lies in a similar kind of branching—instead of assigning one of two truth values to a literal, the branching is based on the valuation trichotomy of atoms:

$$\frac{S}{S \cup \{a = 0\} \mid S \cup \{0 \prec a, a \prec 1\} \mid S \cup \{a = 1\}}$$

for atom a occurring in theory S .

The search-space constraining rules in classical DPLL also have their product-logic counterparts (Guller, 2020); however, in the latter case, the list of applicable rules grows longer: there are thirteen rules necessary for the procedure to be refutation-complete, and four admissible rules that further constrain the search space.

The main idea of using the rules to prove a property (satisfiability, validity) of a theory is similar to that of classical DPLL: the inference starts at the root node of the tree, and the rules are recursively applied to split and simplify the tree. After the traversal, the input theory is satisfiable iff there exists an open branch. In the positive case, the model may be calculated based on the path of traversal.

The papers (Guller, 2013; Guller, 2020) propose and describe the product fuzzy DPLL rules, prove the refutational soundness and completeness of the procedure, explore into detail the results of applying the individual rules, and provide examples of proving the satisfiability and validity of product propositional formulae.

Before we list the rules of the procedure, we recall some of the concepts of (Guller, 2020) in order to better grasp the following notation.

C is a guard iff either $C = a = 0$, $C = 0 \prec a$, $C = a \prec 1$, or $C = a = 1$. Let S be an order propositional clause. We denote $guards(a) = \{a = 0, 0 \prec a, a \prec 1, a = 1\}$ and $guards(S) = \{C \mid C \in S \text{ is a guard}\}$. Atom a is *fully guarded* in theory T iff the theory contains either the literal $a = 0$, the literal $a = 1$, or both of the literals $0 \prec a, a \prec 1$.

Next, we list the definitions of two auxiliary functions. The function *simplify* in Definition 1 modifies an expression by replacing the occurrences of atoms with their truth constants and returns a simplified ex-

pression w.r.t. laws of the standard product algebra.

Definition 1. Auxiliary function *simplify* (Guller, 2020)

$$\begin{aligned} \text{simplify}(0, a, v) &= 0; \\ \text{simplify}(1, a, v) &= 1; \\ \text{simplify}(Cn, a, 0) &= \begin{cases} 0 & \text{if } a \in \text{atoms}(Cn), \\ Cn & \text{else;} \end{cases} \\ \text{simplify}(Cn, a, 1) &= \begin{cases} 1 & \text{if } \exists n^* Cn = a^{n^*}, \\ Cn - a^{n^*} & \text{if } \exists n^* a^{n^*} \in Cn \neq a^{n^*}, \\ Cn & \text{else;} \end{cases} \\ \text{simplify}(l, a, v) &= \text{simplify}(\varepsilon_1, a, v) \diamond \text{simplify}(\varepsilon_2, a, v) \\ & \quad \text{if } l = \varepsilon_1 \diamond \varepsilon_2; \\ \text{simplify}(C, a, v) &= \{\text{simplify}(l, a, v) \mid l \in C\}. \end{aligned}$$

The auxiliary function \odot returns the product of two {conjunctions, literals}. For conjunctions of powers of atoms Cn_1, Cn_2 , literals l_1, l_2 , and expressions ε (truth constants or conjunctions of powers) and μ (truth constants or literals), the function is defined in Definition 2.

Definition 2. Auxiliary function \odot (Guller, 2020)

$$\begin{aligned} 0 \odot \varepsilon &= \varepsilon \odot 0 = 0; \\ 1 \odot \varepsilon &= \varepsilon \odot 1 = \varepsilon; \\ Cn_1 \odot Cn_2 &= \{a^{m+n} \mid a^m \in Cn_1, a^n \in Cn_2\} \cup \\ & \quad \{a^n \mid a^n \in Cn_1, a \notin \text{atoms}(Cn_2)\} \cup \\ & \quad \{a^n \mid a^n \in Cn_2, a \notin \text{atoms}(Cn_1)\} \\ 0 \odot \mu &= \mu \odot 0 = 0; \\ 1 \odot \mu &= \mu \odot 1 = \mu; \\ l_1 \odot l_2 &= (\varepsilon_1 \odot \varepsilon_2) \diamond (v_1 \odot v_2) \text{ if } l_i = \varepsilon_i \diamond v_i, \\ \diamond &= \begin{cases} = & \text{if } \diamond_1 = \diamond_2 = =, \\ < & \text{else.} \end{cases} \end{aligned}$$

For comprehensiveness, below we list the first thirteen fuzzy product DPLL rules as defined in (Guller, 2013; Guller, 2020) and briefly describe their intuition.

(Unit contradiction rule) (1)

$$\frac{S}{S \cup \{\square\}};$$

S is unit;
 there exist $0 < a_0, \dots, 0 < a_m$,
 $a_0 < 1, \dots, a_m < 1 \in \text{guards}(S)$,
 $l_0, \dots, l_n \in S$ such that l_i is pure order literal,
 $\text{atoms}(l_0, \dots, l_n) = \{a_0, \dots, a_m\}$;
 there exist $\alpha_i^* \geq 1, i = 0, \dots, n$,
 $J^* \subseteq \{j \mid j \leq m\}, \beta_j^* \geq 1, j \in J^*$, such that
 $(\odot_{i=0}^n l_i^{\alpha_i^*}) \odot (\odot_{j \in J^*} (a_j < 1)^{\beta_j^*})$ is a contradiction.

Rule (1) derives \square iff a \odot -product of powers of the input pure order literals and guards $a_j < 1, j \in J^*$ can be found that would lead to a contradiction of the form $\varepsilon < \varepsilon$.

(Trichotomy branching rule) (2)

$$\frac{S}{S \cup \{a = 0\} \mid S \cup \{0 < a, a < 1\} \mid S \cup \{a = 1\}};$$

$a \in \text{atoms}(S)$.

The branching rule (2) splits the derivation into three sub-cases of the trichotomy $a = 0 \vee 0 < a \wedge a < 1 \vee a = 1$.

(Pure trichotomy branching rule) (3)

$$\frac{S}{(S - \{\varphi\}) \cup \{l_1\} \mid (S - \{\varphi\}) \cup \{C\} \cup \{l_2\} \mid (S - \{\varphi\}) \cup \{C\} \cup \{l_3\}};$$

$$\varphi = (l_1 \vee C) \in S, C \neq \square,$$

$$l_1 \vee l_2 \vee l_3 \text{ is a pure trichotomy.}$$

Rule (3) is a branching rule splitting the derivation into the three sub-cases of the trichotomy of pure literals l_1, l_2 , and l_3 .

(Contradiction rule) (4)

$$\frac{S}{(S - \{l \vee C\}) \cup \{C\}};$$

$l \vee C \in S, l$ is a contradiction.

The order literal l can be removed from the input order clause $l \vee C$ if it is a contradiction.

(Tautology rule) (5)

$$\frac{S}{S - \{l \vee C\}};$$

$$l \vee C \in S, l \text{ is a tautology.}$$

The input order clause $l \vee C$ can be removed from S if it is a tautology.

(0-simplification rule) (6)

$$\frac{S}{(S - \{C\}) \cup \{\text{simplify}(C, a, 0)\}};$$

$a = 0 \in \text{guards}(S), C \in S$,
 $a \in \text{atoms}(C), a = 0 \neq C$.

If $a = 0 \in \text{guards}(S)$ and the input order clause C contains a , then C can be simplified.

(1-simplification rule) (7)

$$\frac{S}{(S - \{C\}) \cup \{\text{simplify}(C, a, 1)\}};$$

$a = 1 \in \text{guards}(S), C \in S, a \in \text{atoms}(C), a = 1 \neq C.$

If $a = 1 \in \text{guards}(S)$ and the input order clause C contains a , then C can be simplified.

(0-contradiction rule) (8)

$$\frac{S}{(S - \{a_0^{\alpha_0} \& \dots \& a_n^{\alpha_n} = 0 \vee C\}) \cup \{C\}};$$

$0 \prec a_0, \dots, 0 \prec a_n \in \text{guards}(S),$
 $a_0^{\alpha_0} \& \dots \& a_n^{\alpha_n} = 0 \vee C \in S - \text{guards}(S).$

If $0 \prec a_0, \dots, 0 \prec a_n \in \text{guards}(S), a_0^{\alpha_0} \& \dots \& a_n^{\alpha_n} = 0$ is contradictory; it can be removed from the input order clause and C can be derived. The 1-contradiction rule is analogous.

(1-contradiction rule) (9)

$$\frac{S}{(S - \{a_0^{\alpha_0} \& \dots \& a_n^{\alpha_n} = 1 \vee C\}) \cup \{C\}};$$

$a_i \prec 1 \in \text{guards}(S), i \leq n,$
 $a_0^{\alpha_0} \& \dots \& a_n^{\alpha_n} = 1 \vee C \in S - \text{guards}(S).$

(0-consequence rule) (10)

$$\frac{S}{S - \{0 \prec a_0^{\alpha_0} \& \dots \& a_n^{\alpha_n} \vee C\}};$$

$0 \prec a_0, \dots, 0 \prec a_n \in \text{guards}(S),$
 $0 \prec a_0^{\alpha_0} \& \dots \& a_n^{\alpha_n} \vee C \in S - \text{guards}(S).$

If $0 \prec a_0, \dots, 0 \prec a_n \in \text{guards}(S)$, then it must hold that $0 \prec a_0^{\alpha_0} \& \dots \& a_n^{\alpha_n}$. Therefore, the input order clause $0 \prec a_0^{\alpha_0} \& \dots \& a_n^{\alpha_n} \vee C$ is a consequence of the guard(s) and may be removed.

(1-consequence rule) (11)

$$\frac{S}{S - \{a_0^{\alpha_0} \& \dots \& a_n^{\alpha_n} \prec 1 \vee C\}};$$

$a_i \prec 1 \in \text{guards}(S),$
 $i \leq n, a_0^{\alpha_0} \& \dots \& a_n^{\alpha_n} \prec 1 \vee C \in S - \text{guards}(S).$

Analogous to rule (10).

(0-annihilation rule) (12)

$$\frac{S}{S - \{a = 0\}};$$

$a = 0 \in \text{guards}(S), a \notin \text{atoms}(S - \{a = 0\}).$

(1-annihilation rule) (13)

$$\frac{S}{S - \{a = 1\}};$$

$a = 1 \in \text{guards}(S), a \notin \text{atoms}(S - \{a = 1\}).$

If the atom a different from 0, 1 occurs in S only in the guard $a = 0$ or $a = 1$, then this guard may be removed from S .

In our previous work (Uhliarik, 2019) we proposed a deterministic algorithm that uses these rules to split and traverse the tree in an informed fashion; however, it was still left open how to effectively accomplish the application of the unit contradiction rule. The contribution of this paper lies in (i) the revision of the algorithm, (ii) the proposal of an approach to detect contradiction within the unit contradiction rule, and (iii) the description and evaluation of our concrete, working implementation of the solver. The next section deals with (i) and (ii).

3 ALGORITHM

Given only the list of branching rules, a naïve tree-traversing algorithm would explore the whole search space to determine the existence of a closed branch. Simplification rules may help constrain the traversal, but without a specification directing how to use these rules, the naïve algorithm could apply them eagerly, which could lead to premature branching and inefficient traversal.

For this reason we proposed an algorithm in our previous work (Uhliarik, 2019) that constructs and traverses the product DPLL tree in order to prove the (non-)satisfiability of an order clausal theory. The algorithm uses the branching and simplifying rules of the product fuzzy DPLL procedure (Guller, 2013) in an informed way, based on our analysis of branch properties after the application of individual rules.

However, during our later work on the implementation and testing of the solver, we discovered that the traversal could be enhanced to further constrain the search space. In this section we introduce the revision of the algorithm. In addition to modifications, we try

to be more specific about the more complex steps of the procedure.

3.1 Translation into Order Clausal Form

We recall that the DPLL procedure constructs a tree from the (non-empty) product order clausal theory. The algorithm to translate any such theory into the order clausal form is based on the application of interpolation rules introduced in (Guller, 2013, Sect. 3) and shown in Alg. 1. For simplicity we consider the input to be a single (complex) order propositional formula—in the case of a set of formulae, the procedure yields intuitive results for the \wedge -conjunction of its elements in Π_Δ without loss of generality. While traversing the structure of the formula (represented as a tree), its subformulae are matched by interpolation rules and substituted with generated auxiliary atoms.

The crucial change is in line 3 of the algorithm. Previously we considered the overall solver to prove only the satisfiability of the input formula. For this, the first step of the translation of formula φ was the addition of the literal $\tilde{a}_0 = 1$, \tilde{a}_0 being the auxiliary atom $\varphi \leftrightarrow \tilde{a}_0$. To enable checking for validity, the first literal becomes $\tilde{a}_0 < 1$ (and the result is assumed negated).

The rest of the algorithm performs the standard pre-order traversal of the formula structure. In every step, the pair of subformula ψ and its associated auxiliary atom $\tilde{a}_i \leftrightarrow \psi$ is picked from the queue. The function $\text{Inter}(\tilde{a}_i, \psi)$ (omitted for brevity) processes the pair by using an interpolation rule matching ψ according to (Guller, 2013; Guller, 2020) and yields two sets: the set $nPairs$ of pairs of new auxiliary atoms and subformulae of ψ , and the new clauses $nClauses$ resulting from applying the interpolation rule. The pairs $nPairs$ are further processed in next iterations, and the clauses $nClauses$ are added to the result.

For a visualization of performing the translation, refer to (Guller, 2013, Table 6).

3.2 DPLL Inference

The algorithm accepts as input an order clausal theory and uses the rules (1)–(13) to split the tree into branches and simplify the theory. A branch is closed when the empty clause \square is derived in its leaf, otherwise the branch is open. A tree is closed iff all its branches are closed, otherwise it is open. The theory is satisfiable iff an open branch exists after the exhaustion of applicable rules (Guller, 2013, Theorem 4.2).

Input: Formula φ

Result: Order clausal theory S^φ

```

1 Function Translate( $\varphi$ ):
2    $queue \leftarrow [(\tilde{a}_0, \varphi)]$ ;
3    $S^\varphi \leftarrow \{\tilde{a}_0 = 1\}$  (SAT) or  $\{\tilde{a}_0 < 1\}$  (VAL);
4   while  $queue$  not empty do
5      $\tilde{a}_i, \psi \leftarrow$  pick pair from  $queue$ ;
6      $nPairs, nClauses \leftarrow$  Inter( $\tilde{a}_i, \psi$ );
7     foreach  $pair \in nPairs$  do
8       | add  $pair$  to  $queue$ ;
9     end
10    foreach  $clause \in nClauses$  do
11      | add  $clause$  to  $S^\varphi$ ;
12    end
13  end
14  return  $S^\varphi$ 
    
```

Algorithm 1: The translation of order propositional formula into order clausal theory.

The intuition of the algorithm is depicted in a simple form by the flowchart in Figure 1 and described more precisely by pseudo-code in Algs. 2–6.

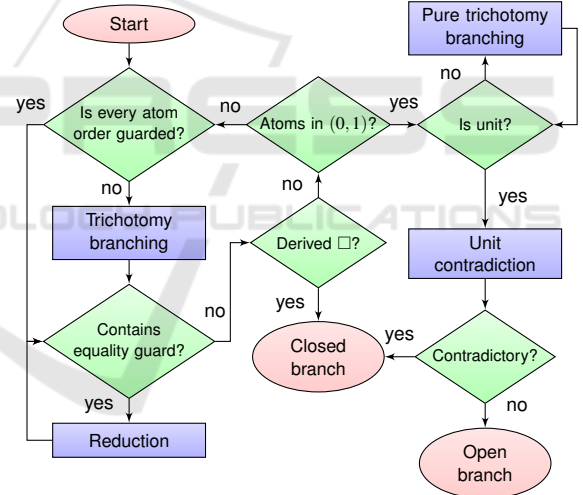


Figure 1: Flowchart of the inference algorithm performing the DPLL procedure.

The first steps depicted in Algs. 2 and 3 are the same as in our previous work (Uhlirik, 2019), with the exception that we will not concern ourselves with finding the model related to an open branch. A non-empty input theory S is first processed by the initial call to the $\text{Trichotomy}(S)$ function (Alg. 2). The function creates branches recursively according to the content of S by using the trichotomy branching rule (2) as shown in Alg. 3. For each atom in S , any missing order guards of the atom to make it fully guarded ($a = 0$; $0 < a, a < 1$; $a = 1$) are added to the theory in the subsequent branches. Each branch $s \in S$

is then simplified using the `Reduce(s)` function and `true` is returned (S is satisfiable) if `Reduce(S)` returns `true` (the branch could not be recursively closed). If all branches have been closed, return `false`. Otherwise we assume all atoms to be fully guarded and delegate satisfiability checking to the reduction of S .

Input: A non-empty order clausal theory S

Result: `true` iff S is satisfiable

1 **return** `Trichotomy(S)`;

Algorithm 2: The initial step of the DPLL procedure.

```

1 Function Trichotomy(S) :
2   foreach  $a \in \text{atoms}(S)$  where  $a$  is not fully
   guarded do
3      $S' \leftarrow$  split  $S$  using rule (2) over  $a$ ;
4     foreach  $s \in S'$  do
5        $s \leftarrow$  Reduce(s);
6       if  $s \neq \square$  then
7         return true;
8       end
9     end
10    if all  $s \in S'$  are closed then
11      return  $\square$ ;
12    end
13  end
14  return Reduce(S);
```

Algorithm 3: The `Trichotomy` function of the DPLL procedure.

The task of the `Reduce(S)` function is to apply the simplification rules (4)–(13) that reduce the theory by eliminating unnecessary clauses and literals. The rules are applied consecutively until all equality guards are resolved, returning `false` immediately after a closed branch is derived. Then the checking of satisfiability is delegated to the `PureTrichotomy(S)` function if all atoms are fully guarded, or the missing guards are supplemented by the `Trichotomy(S)` function. This description contains a modification in contrast to (Uhlirak, 2019) marked with (*): in addition to checking whether the reduction rules derive \square , it is important to handle the situation when the rules (5) and (10)–(13) eliminate all clauses of the theory. In such a case, the branch is open and the function returns `true`.

The function `PureTrichotomy(S)` is described in Alg. 5. To ensure all equality guards have been cleared in S , we repeat the use of `Reduce(S)` at this level of the tree. Afterward, the input theory is guaranteed to be pure and fully guarded. We handle unit theories by passing them to `UnitContradiction(S)` and return the negated result, as the function returns `true` iff the theory is contradictory. Otherwise we pro-

```

1 Function Reduce(S) :
2   while  $S$  contains equality guard do
3      $S \leftarrow$  Application of rules (4)–(13)
       consecutively. Return false if any of
       the rules returns false. Return true
       if any of the rules (5), (10)–(13)
       return true (*);
4   end
5   if all atoms in  $S$  are fully guarded then
6     return PureTrichotomy(S);
7   end
8   return Trichotomy(S);
```

Algorithm 4: The `Reduce` function of the DPLL procedure.

ceed to the application of the pure trichotomy branching rule (3). The function iterates over every non-unit clause and attempts to create at most three branches. For the literal a , alternative literals b and c are generated that form the pure trichotomy $a \vee b \vee c^2$. The branching rule dictates that we create at most three branches according to the pure trichotomy in question (in each branch, only one of the literals is true). If we proceeded to apply the branching rule (line 18) by modifying only the clause at hand, the branching would continue until all literals in all clauses have been processed by `UnitContradiction(S)`. Therefore, we remove every occurrence of the counterpart literals in S at this step, which significantly reduces the search space. Before proceeding to create the branches, we also use the simple look-ahead technique (lines 7–17) to check whether any of the literals a , b , or c already appear in the theory and assume their respective truth. In contrast to (Uhlirak, 2019), this constrains the search space even further, as many pure literal contradictions are eliminated at this level.

If a closed branch is derived after the removal of counterpart literals, we return `false`, otherwise we recursively call the `PureTrichotomy` function for every branch. The iteration over literals in clauses is unnecessary, as the branching here already modifies the branched theories, so that in the worst case, every literal is checked once.

The last function `UnitContradiction(S)` shown in Alg. 6 uses the unit contradiction rule (1). For theory S , the function returns `true` iff there exists a contradictory \odot -product of powers of literals, in which case the branch is closed.

Note that at this step, if the branch remains open, the theory is satisfiable and a valuation can be found, e.g. using the partial valuation method described in (Guller, 2013, Table 5). However, the current state

²E.g. the concrete literal $x \prec y$ generates the counterpart literals $y \prec x, x = y$.

```

1 Function PureTrichotomy( $S$ ):
2   if  $S$  contains equality guard then return
   Reduce( $S$ );
3   if  $S$  is unit then return
   UnitContradiction( $S$ ) == false;
4    $S' \leftarrow S$  without unit clauses;
5   foreach clause  $c \in S'$  do
6      $a \leftarrow$  first literal in  $c$ ;
7     if  $a$  is in any unit clause of  $S$  then
8       remove clauses containing  $a$  in  $S$ ;
9       add the unit clause  $\{a\}$  to  $S$ ;
10      return PureTrichotomy( $S$ );
11    end
12     $b, c \leftarrow$  generated literals forming pure
    trichotomy  $a \vee b \vee c$ ;
13    if  $b$  or  $c$  is in any unit clause of  $S$  then
14      remove every occurrence of literal
       $a$  from clauses in  $S$ ;
15      if derived  $\square$  then return false;
16      return PureTrichotomy( $S$ );
17    end
18     $S_1, S_2, S_3 \leftarrow$  split  $S$  using rule (3) over
     $a, b, c$ , removing occurrences of
    negative literals from all clauses;
19    if derived  $\square$  in any of  $S_1, S_2, S_3$  then
    return false;
20    foreach  $S'' \in \{S_1, S_2, S_3\}$  do
21      if PureTrichotomy( $S''$ ) then
22        return true;
23      end
24    return false;
25  end

```

Algorithm 5: The PureTrichotomy function of the DPLL procedure.

```

1 Function UnitContradiction( $S$ ):
2   if  $S$  is unit and rule (1) is applicable then
3     return true;
4   else
5     return false;
6   end

```

Algorithm 6: The UnitContradiction function of the DPLL procedure.

of the solver and the scope of this paper does not cover model-finding.

3.3 Unit Contradiction

In our previous work (Uhliarik, 2019) we described the task of the unit contradiction rule (1) and analyzed the approach of applying the rule on input theories.

The paper proposed an encoding of the theory as a linear programming instance, but has not reached a conclusion on how to find the solution to the problem. In addition to the enhancements of the algorithm performing DPLL described in Section 3.2, we fill the gap by proposing a method to solve this part of the procedure.

The applicability of the unit contradiction rule on theory S is the problem of finding a contradictory product of the form $\varepsilon \prec \varepsilon$ in S by using the operation \odot over pure order literals and the strict order guards $a_i \prec 1$. In other words, it is the task of choosing powers of literals such that their \odot -product yields the contradiction $\varepsilon \prec \varepsilon$.

Example 1 (Uhliarik, 2019) illustrates this problem on the theory with fully guarded atoms a, b , and the pure order literals $\{a^2 = b^3, b \prec a\}$. The boxed literals (14), (15), (16) are associated with the respective powers 1, 2, 1. By the \odot -multiplication of these powers of literals we obtain the contradiction $a^2 \& b^3 \prec a^2 \& b^3$ (17).

Example 1. Application of the unit contradiction rule (Uhliarik, 2019)

$$0 \prec a, 0 \prec b, a \prec 1, \boxed{b \prec 1}$$

$$\boxed{a^2 = b^3}, \boxed{b \prec a}$$

$$a^2 = b^3 \quad (14)$$

$$(b \prec a)^2 \quad (15)$$

$$b \prec 1 \quad (16)$$

$$a^2 \& b^3 \prec a^2 \& b^3 \text{ — a contradiction} \quad (17)$$

Remark 1. To obtain a contradictory product, at least one of the literals has to be strict order, according to Definition 2 (operator \odot).

This problem can be solved using linear programming (LP) with the representation of problems following our previous work (Uhliarik, 2019). For a unit order clausal theory, let the variables of the LP problem represent its pure order literals and the guards $a_i \prec 1$, and the constraints represent the atoms. To handle the commutativity of the equality operator $=$, we encode each equality literal in the matrix twice, once with the operands reversed. Then, let the coefficients of the variables be equal to the difference of powers of atoms appearing on the left- and right-side of literals³. In the canonical form of linear programs $Ax = b$, the matrix A is shown in Eq. 18,

³E.g. given the literal $a^2 \& b^3 \prec b^2 \& c$, the coefficient for atom a is $2 - 0 = 2$, for atom b it is $3 - 2 = 1$, and for atom c it is $0 - 1 = -1$.

$$A = \begin{bmatrix} a_{1,1} & \cdots & a_{1,2q} & \cdots & a_{1,2q+p} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{m,1} & \cdots & a_{m,2q} & \cdots & a_{m,2q+p} \end{bmatrix} \quad (18)$$

where the rows represent atoms, columns the literals; $a_{i,j}$ being the difference of the power assigned to atom at position i in literal at position j on the left and right side of the literal; q is the number of equality literals, p is the number of strict order literals (Uhliarik, 2019).

For the theory in Example 1 the coefficient matrix is as shown in Eq. 19.

$$A_{ex} = \begin{bmatrix} 2 & -2 & -1 & 1 & 0 \\ -3 & 3 & 1 & 0 & 1 \end{bmatrix} \quad (19)$$

The choice of powers of literals used in the operation \odot must be such that in the resulting form $\varepsilon \prec \varepsilon$ the difference of powers of atoms on the left- and right-hand side are equal—the value of atoms in the \odot -product of literals must be equal to 0. Hence, we consider the LP constraint bounds to be the vector b in Eq. 20,

$$b = \overbrace{[0, \dots, 0]}^m \quad (20)$$

where m is the number of atoms.

Since we assume any literal to be eligible with the same weight, we represent the linear programming objective function coefficients by the vector c shown in Eq. 21,

$$c = \overbrace{[1, \dots, 1]}^{2q+p} \quad (21)$$

where q is the number of equality literals and p is the number of strict order literals.

Finally, we seek to minimize the powers of literals, so we assume the minimization objective.

With this configuration, however, the zero vector is always the minimal solution for any theory, meaning the solution is such where no literal is used in the \odot -product. This obviously cannot yield a contradiction. To ensure we reject such solutions, we add an additional constraint to only accept solutions such that $x \geq 1$. The matrix of coefficients A and the vector of constraint bounds b then become A' (22) and b' (23),

$$A' = \begin{bmatrix} a_{1,1} & \cdots & a_{1,2q} & \cdots & a_{1,2q+p} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{m,1} & \cdots & a_{m,2q} & \cdots & a_{m,2q+p} \\ 1 & \cdots & \cdots & \cdots & 1 \end{bmatrix} \quad (22)$$

$$b' = \overbrace{[0, \dots, 0]}^{m+1} \quad (23)$$

where m is the number of atoms, q the number of equality literals and p the number of strict order literals. The type of constraint remains = (equality) for the first m constraints, and changes to \geq for the additional constraint.

As the variables in linear programming are by nature continuous, the values of the solution might not be integers. For the literals in Example 1, the solution might be one such that the obtained powers of literals (14), (15), (16) are 0.5, 1, and 0.5. This is equally acceptable, as the intuitive interpretation using integers can be calculated by multiplying each power with their least common multiple.

Remark 2. *Throughout this paper we argue that one of the advantages of our approach to solving satisfiability in product propositional logic is its non-reliance on translations into other systems, but our method of handling unit contradiction does not uphold to this statement. While this is true, our use of an external linear programming solver is isolated to the application of the unit contradiction rule. It is not necessarily invoked in every problem instance. Despite this, we find the use of an LP solver excessive and are considering a simpler, more direct approach for future development.*

4 IMPLEMENTATION

We have developed a working implementation of the solver according to the algorithm proposed in Section 3, which is available for download⁴ and free to use⁵.

The main software artifact `prodfsat` is a console application that parses an input file containing product propositional theories and outputs for each whether the theory (a) is satisfiable and (b) its negation is satisfiable (to determine whether it is valid).

The solver was developed using test-driven development and the executable test suites are available in the artifact `prodfsat_tests`. In addition to unit tests, the solver is tested on a suite of sample theories consisting of axioms and properties from (Hájek, 2001), (Guller, 2020), and a few custom examples.

In the rest of this section we describe the usage of the main application and its features and limitations, demonstrate the application on an example, and provide additional technical details.

⁴<https://git.uhliarik.com/ivor/prodfsat>

⁵Under the GNU General Public License v3.0 or later.

4.1 Application Usage

The console application accepts arguments according to the following specification:

```
prodfsat [-s] [-p] [FILE]...
```

where:

- the switches `-s`, `-p` enable the output of debugging information related to input scanning and parsing,
- the list of positional arguments `FILE...` are the files containing the input product propositional theories.

The input is a plain-text file. The syntax the program accepts is designed to be intuitive and versatile, and follows the syntax and connective precedence of Π_{Δ} with the mapping shown in Table 1. Additionally, atoms consist of alphanumeric characters, where the first character must be alphabetic. The constants 0, 1 are straightforward, and powers of atoms must be positive integers. To override operator precedence, the use of parentheses is supported as in the expression `"(-a && -b) == -(a || b)"`. The input may contain a comment of the form `"# ..."` which is skipped during parsing, but may only appear at the beginning of a line. The appearance of two or more consecutive newline characters acts as the separator of theories. Note that the Δ connective is not supported; its use as in the expression Δx must be substituted with $x = 1$.

In Section 3.1 we've stated that we represent the input theory as a single formula produced by the \wedge -conjunction of the assumed theory's elements. For this reason, the last row of Table 1 lists the connective \wedge again, but instead with the lowest precedence; commas and (single) newlines are treated as this case of \wedge .

Table 1: The mapping of text strings to logical expressions.

Expression	Text string	Example
power	"^"	"a^3"
\neg	"_"	"-a"
&	"&"	"a & 0"
=	"="	"a = 0"
\prec	"<"	"0 < a"
\wedge	"&&"	"0<a && a<1"
\vee	"v", "V", " "	"-a V --a"
\rightarrow	"->", "-:"	"a=1 -> a=0"
\leftrightarrow	"<->", "=="	"a&b == aVb"
\wedge (minprec)	",", "\n	"a<1, b<1"

We have pointed out throughout the paper that our solver does not yet find the models in the sense of the valuation of atoms. Any mentions of the construction of models in the source code at the time of writing instead refer to the listing of literals that are true in an open branch.

4.2 Example

To demonstrate the application, we present an example of proving the validity of the formula (24).

$$(0 \prec c) \& (a \& c \prec b \& c) \rightarrow a \prec b. \quad (24)$$

We have encoded this formula into the text file `formula.txt` in the following form:

```
(0 < c) & (a & c < b & c) -> a < b
```

Now we run the program using the filename as the argument: `prodfsat formula.txt`

The application first informs us about the successful parsing of the formula:

```
[info] Parsing result:
((0 < c & (a & c) < (b & c)) -> a < b)
```

Then, the program translates the formula into order clausal form for solving the SAT problem and outputs this form, where each auxiliary atom is named with an asterisk and its order number. The same is written out for the VAL case (not shown).⁶

```
[info] SAT: In clausal form:
(((*0) = (1)))
(((*1) < (*2)) V ((*1) = (*2))
...V ((*1 & *0) = (*2)))
(((*2) < (*1)) V ((*0) = (1)))
(((*1) = (*3 & *4)))
(((*5) < (*6)) V ((*2) = (0)))
(((*6) < (*5)) V ((*6) = (*5))
...V ((*2) = (1)))
((0 < (*7)) V ((*3) = (0)))
(((*7) = (0)) V ((*3) = (1)))
(((*8) < (*9)) V ((*4) = (0)))
(((*9) < (*8)) V ((*9) = (*8))
...V ((*4) = (1)))
(((*5) = (a)))
(((*6) = (b)))
(((*7) = (c)))
(((*8) = (*10 & *11)))
(((*9) = (*12 & *13)))
(((*10) = (a)))
(((*11) = (c)))
(((*12) = (b)))
(((*13) = (c)))
```

⁶The output of the program listed here has been aesthetically modified.

Finally, the application outputs the results of SAT and VAL testing.

```
[info] Satisfiability: true
[info] Satisfied literals:
(*12) = (0)
(*11) = (0)
(*10) = (0)
(*1) = (0)
[info] Satisfiability of negation: false
```

We find that the formula is satisfiable (if every atom has the value 0, the antecedent is false) and its \neg -negation is unsatisfiable (the formula is valid).

4.3 Technical Details

The application is implemented in the C++ language. In the source code we utilize features of modern C++ standards up to C++20, for the purpose of straightforward representation (structured binding declaration of tuple elements, list initialization), and to achieve efficient memory handling (move semantics, mandatory copy elision).

While the implementation strives to be operating-system agnostic, it has only been tested in the Linux environment. The complete list of build and runtime dependencies, as well as the instructions to build the project are documented on the project's website.

5 EVALUATION

To validate the correctness of the implementation and evaluate its performance, we have created the set of 69 test formulae originating in fuzzy logic literature (Hájek, 2001; Guller, 2020) and some of our custom input. The formulae comprise the axioms of Hájek's basic logic (8 formulae), the properties of basic logic (49 formulae), product logic axioms and formulae proved in Π (5 formulae); two examples from Guller, and five examples created during development. The tests formulated as inputs for our solver can be found among the project's source code, and the tests may be executed by running the `prodfsat_tests` program (test suite Solving).

For every formula, both the satisfiability and validity checks have been performed. The samples from (Hájek, 2001) should all be tautologies, others are either tautologies, only satisfiable, or unsatisfiable.

We have grouped the samples by the area of their origin (e.g. BLConj consists of the properties of conjunction in basic logic), which also reflects their separation inside the project's test files, and measured

the runtime⁷ required to prove (non-)satisfiability and (non-)validity. To conduct the measuring, the executable `prodfsat_tests` was built using the default release profile of the CMake build system version 3.18.0 and the compiler GCC version 10.1.0. The measurements were recorded five times, each time after clearing the system's cache. The results in milliseconds are provided in Table 2, where SD stands for standard deviation and NM is the mean normalized by the size of group.

Table 2: Measurement of runtime of test samples in milliseconds.

Group	Size	Mean	SD	NM
BLAxioms	8	709.2	43.51	88.65
BLConj	8	966.4	35.75	120.80
BLConstant	3	4.4	0.55	1.47
BLDisj	8	953.2	12.32	119.15
BLEquiv	9	1017.2	15.56	113.02
BLImpl	3	107.0	2.55	35.67
BLNegation	6	54.6	3.44	9.10
BLStrongConj	6	706.2	12.83	117.70
BLMisc	6	519.2	25.17	86.53
ProductLogic	5	115.4	2.51	23.08
Guller	2	234.6	4.56	117.30
Custom	5	135.2	6.14	27.04
Total	69	5587.4	90.03	80.98

As can be seen in Table 2, the average time of running the full suite of samples is 5.59 seconds, and the average time to run a single pair of SAT and VAL tests is 80.98 ms. The measurements of individual test samples (omitted for brevity) would reveal that the minimum runtime was in some cases (BL axiom 2 and 8, BL constant samples, etc.) as low as only a few hundred microseconds.

We have yet to establish a set of benchmark tests to be able to perform comparative empirical evaluation with other existing approaches. The construction of such a benchmark is considered for future work. The MNiBLoS solver (Vidal, 2016) mentioned in Section 1 is the closest to our solution in its goals, but for now, only the small intersection of the results on BL axiom samples may be discussed.

In (Vidal, 2016, Sect. 4.2), the first set of measurements has been conducted over the product generalizations of BL axioms of the form $(p^n \& q^n) \rightarrow p^n$.

⁷The measurements were conducted on a personal computer with the CPU frequency of 3.31 GHz; the implementation is single-threaded.

With increasing n , it is shown that the runtime increases polynomially. In these cases, our solution is advantageous in the representation of powers of atoms, as the increase of n does not change the behavior of the solver, and the average runtime stays constant. The runtime of MNiBLoS for these formulae over different values of n is on average higher than ours (our solution having the average runtime of 21 ms with BL axiom 3 and 119 ms with BL axiom 4), but these differences are not representative, as they were conducted on different machines, and the author of MNiBLoS proposes a better set of tests that has yet to be integrated and compared with our solution.

6 CONCLUSION AND FUTURE WORK

In this paper we have presented the results of our work, which included the refinement of our deterministic version of the product fuzzy DPLL algorithm and proposed a method of applying the unit contradiction rule. Also, we have reached the milestone of our research by contributing the first working implementation of the solver of its kind.

However, there are still areas that could be improved with future work. First, we seek to establish a benchmark for performing more extensive comparative evaluation with existing solutions. Moreover, we would like to implement the admissible DPLL rules to make the implementation yet more efficient. In addition to solving the SAT and VAL problems, in the future versions of the implementation we will also attempt to integrate solving the deduction (DED) problem and finding valuations for models.

ACKNOWLEDGEMENTS

This work was supported by the Slovak Research and Development Agency under contract No. APVV-19-0220 (ORBIS) and by the Slovak VEGA agency under contract No. 1/0778/18 (KATO).

REFERENCES

- Alviano, M. and Peñaloza, R. (2015). Fuzzy answer set computation via satisfiability modulo theories. *TPLP*, 15(4-5):588–603.
- Baaz, M., Hájek, P., Švejda, D., and Krajíček, J. (1998). Embedding logics into product logic. *Studia Logica*, 61(1):35–47.
- Béjar, R., Alsinet, T., Bou, F., Barroso, D., Cerami, M., and Esteva, F. (2013). On the Implementation of a Fuzzy DL Solver over Infinite-Valued Product Logic with SMT Solvers. volume 8078.
- Bobillo, F. and Straccia, U. (2007). A fuzzy description logic with product t-norm. In *2007 IEEE International Fuzzy Systems Conference*, pages 1–6.
- Brys, T., Drugan, M. M., Bosman, P. A., De Cock, M., and Nowé, A. (2013). Solving Satisfiability in Fuzzy Logics by Mixing CMA-ES. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation, GECCO '13*, pages 1125–1132, New York, NY, USA. ACM.
- Davis, M., Logemann, G., and Loveland, D. (1962). A Machine Program for Theorem-proving. *Commun. ACM*, 5(7):394–397.
- Guller, D. (2013). A DPLL procedure for the propositional product logic. In *Proceedings of the 5th International Joint Conference on Computational Intelligence - Volume 1: FCTA, (IJCCI 2013)*, pages 213–224. INSTICC, SciTePress.
- Guller, D. (2020). Technical Foundations of a DPLL Procedure for Propositional Product Logic. Submitted manuscript in *Journal of Applied Logics – IfCoLog Journal of Logics and their Applications*.
- Hájek, P. (2001). *Metamathematics of Fuzzy Logic*. Trends in Logic. Springer.
- Janssen, J., Schockaert, S., Vermeir, D., and De Cock, M. (2012). *Answer Set Programming for Continuous Domains: A Fuzzy Logic Approach*. Atlantis Computational Intelligence Systems. Atlantis Press.
- Mostert, P. S. and Shields, A. L. (1957). On the structure of semigroups on a compact manifold with boundary. *Annals of Mathematics*, 65(1):117–143.
- Suttner, C. and Ertel, W. (1990). Automatic acquisition of search guiding heuristics. In Stickel, M. E., editor, *10th International Conference on Automated Deduction*, pages 470–484, Berlin, Heidelberg. Springer Berlin Heidelberg.
- Uhliarík, I. (2019). *Foundations of a DPLL-Based Solver for Fuzzy Answer Set Programs*, pages 99–117. Springer International Publishing, Cham.
- Vidal, A. (2016). MNiBLoS: A SMT-based solver for continuous t-norm based logics and some of their modal expansions. *Information Sciences*, 372:709 – 730.