SN Computer Science (2022) 3: 388
https://doi.org/10.1007/s42979-022-01192-z

ORIGINAL RESEARCH

Enhancing and Evaluating the Product Fuzzy DPLL Solver

Ivor Uhliarik[1]

Received: 10 August 2021 / Accepted: 4 May 2022 / Published online: 21 July 2022

Abstract
In recent years we have seen a number of satisfiability solvers emerge in the world of fuzzy propositional logics. However, only a few of them can solve product logic problems with the continuous product t-norm. The existing solvers may be categorized into those based on (1) translations into instances of other systems such as satisfiability modulo theories, (2) evolutionary algorithms, and (3) fuzzy generalizations of classical-logic procedures, such as hyperresolution or the Davis–Putnam–Logemann–Loveland (DPLL) procedure. In our previous work we have designed and developed a fuzzy DPLL solver for Δ-extended product propositional logic. This paper presents the enhancements we have made to the previous iteration and establishes a set of experiments motivated by existing solutions for comparing the solver among its versions as well as with other methods. We have conducted the experiments using our solver and two existing solutions. The results show that our solution excels at small inputs and formulae with certain properties. Eventually, we demonstrate the extensibility of our solver by devising an ad hoc simplification rule that compacts the search space in a specific scenario.

Keywords Fuzzy logic · Product logic · Automated theorem proving · DPLL · Satisfiability solving

Introduction

The area of automated theorem proving (ATP) in fuzzy logics has been well-developed over the last few decades in propositional logic (see e.g. Ansótegui et al. [4], Brys at al. [8], Guller [12, 15, 16], Hähnle [18], Vidal [30, 31]), description logics [6, 9, 22], or fuzzy answer set programming [2, 21, 29]. However, among the three prominent (Gödel, Łukasiewicz, and product) logics, only a few of them pay attention to product logic.

In propositional fuzzy logic, ATP relates to the tasks of verifying satisfiability (SAT) and validity (tautologicity, VAL) of formulae. Several papers have proposed approaches that are unified across the three prominent t-norms [4, 8, 30, 31] and as such may benefit from using the Mostert-Shields theorem [23] to support all continuous t-norms.

We highlight the work of Vidal who has developed two such solvers, namely the Nice BL-Logics Solver NiBLoS [31] and its modal extension mNiBLoS [30]. These solvers are capable of checking satisfiability, validity, and deducibility (verifying whether a formula follows from a theory) of formulae. They are based on translations of inputs into satisfiability modulo theory (SMT) instances. SMT solvers are generalizations of SAT solvers with the ability to solve multitude of problems in various domains using so-called background theories (e.g., using the background theories of linear real arithmetic). The NiBLoS solver is based on one of the first such attempts [3] which introduced the straightforward encoding of Łukasiewicz and product logic connectives in SMT by defining the respective fuzzy logic operations as functions that SMT understands. NiBLoS also implements such encodings and adds the support for Gödel logic and any other continuous t-norm-based logic. Once the input formula is translated, it is fed into an SMT solver (such as Intel's Z3 solver [11]) together with the SMT encoding of the respective logic and the result is interpreted to find whether a model exists. The mNiBLoS solver is more advanced than NiBLoS in several ways, the most notable of which are two: the support for modal fuzzy logic and the ability to avoid algebraic multiplication of the product t-norm [30, Sect. 3.1.2]. Both of these solvers have been

This article is part of the topical collection "Computational Intelligence" guest edited by Kurosh Madani, Kevin Warwick, Juan Julian Merelo, Thomas Bäck and Anna Kononova.

✉ Ivor Uhliarik
ivor.uhliarik@uniba.sk

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

validated, empirically tested, and have implementations available to the public.

Another group of approaches to solving the SAT problem in propositional logic includes the work of Guller on the hyperresolution principle [13, 14, 16] and the fuzzy generalization of the Davis–Putnam–Logemann–Loveland (DPLL) procedure [12, 15]. The papers prove the soundness and completeness of the approaches and provide a foundation for developing a solver. Using these methods, such a solver would not have to rely on the translation into other systems and the existence of other solvers.

In our work we focus on SAT solving in product fuzzy logic based on the fuzzy generalization of the DPLL procedure [12], which allows us to develop a self-contained and transparent solver. The specialization on product logic allows us to easily extend the algorithm with simplification rules. Moreover, we choose to work with product logic extended with the Monteiro-Baaz Δ connective (Π_Δ), motivated by the embeddability of Łukasiewicz and Δ-extended Gödel logics within Π_Δ, which allow us to develop a uniform solver for all three prominent fuzzy logics in the future.

This paper is an extension of our previous work [28] where we have proposed a deterministic algorithm to solve SAT and VAL for propositional formulae in Π_Δ and provided details about our working implementation. In this paper we (1) introduce enhancements of the algorithm and describe the improvements made in our implementation, (2) experimentally evaluate its performance and compare the current version with the previous state of our work, and most notably (3) perform experiments for comparative testing of our solution, analyze the results, and compare our solution with the NiBLoS and mNiBLoS solvers. Moreover, we have added a hypothetical motivational example and provided examples to many concepts and notions to make them more intuitive.

In the following sections we first recall the preliminary notions and the fuzzy DPLL procedure ("Preliminaries" section) and demonstrate a possible application of product propositional logic SAT solving on a hypothetical real-world example ("Motivational Example" section). Then, we describe the algorithms used by our solver ("Algorithm" section), describe our implementation ("Implementation" section), and define the experiments and report the results of comparison ("Experimental Results" section). Finally, we conclude the paper ("Conclusion and Future Work" section).

Preliminaries

This section introduces the preliminary notions used throughout the rest of the paper. First, we define the Δ-extended product propositional logic. Then, we discuss the order clausal form of product propositional formulae. Finally, we outline the product fuzzy DPLL procedure.

Table 1 Connectives and operators of Π_Δ

Connective	Operator	Name
$\neg$	$\boldsymbol{\neg}$	Negation
$\wedge$	$\boldsymbol{\wedge}$	(Weak) conjunction
$\&$	$\cdot$	Strong conjunction
$\vee$	$\boldsymbol{\vee}$	Disjunction
$\rightarrow$	$\boldsymbol{\Rightarrow}$	Implication
$\leftrightarrow$	(none)	Equivalence
$\eqcirc$	$\boldsymbol{\eqcirc}$	Equality
$\prec$	$\boldsymbol{\prec}$	Strict order
Δ	$\boldsymbol{\Delta}$	Delta

Product Propositional Logic

The target fuzzy logic of this work is the product propositional logic Π_Δ extended with the Monteiro-Baaz Δ connective and the connectives $\eqcirc$, $\prec$. The logic is interpreted by the product algebra and the related operators $\boldsymbol{\eqcirc}$, $\boldsymbol{\prec}$, and $\boldsymbol{\Delta}$:

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

The syntactical connectives and associated semantic operators of Π_Δ are listed in Table 1 with the decreasing precedence: $(\neg, \Delta, \&, \eqcirc, \prec, \wedge, \vee, \rightarrow, \leftrightarrow)$.

The operations are defined for the operands $x, y \in [0,1]$ with the result in $[0,1]$ as follows:

$$x \boldsymbol{\Rightarrow} y = \begin{cases} 1 & \text{if } x \leq y, \\ \frac{y}{x} & \text{else;} \end{cases} \qquad \boldsymbol{\neg} x = \begin{cases} 1 & \text{if } x = 0, \\ 0 & \text{else;} \end{cases}$$

$$x \boldsymbol{\eqcirc} y = \begin{cases} 1 & \text{if } x = y, \\ 0 & \text{else;} \end{cases} \qquad x \boldsymbol{\prec} y = \begin{cases} 1 & \text{if } x < y, \\ 0 & \text{else;} \end{cases}$$

$$\boldsymbol{\Delta} x = \begin{cases} 1 & \text{if } x = 1, \\ 0 & \text{else.} \end{cases}$$

Next, the operators $\boldsymbol{\vee}$, $\boldsymbol{\wedge}$ are defined as the supremum and infimum operator on $[0,1]$, respectively; $\cdot$ as the algebraic product; the equivalence[1] connective in the expression $x \leftrightarrow y$ as $x \boldsymbol{\Rightarrow} y \boldsymbol{\wedge} y \boldsymbol{\Rightarrow} x$. The algebra's absorbing and neutral elements 0 and 1 are the interpretations of the truth constant of absolute falsehood and absolute truth.

The residuum operator $\boldsymbol{\Rightarrow}$ satisfies the residuation principle w.r.t. operator $\cdot$. For any $x \in \boldsymbol{\Pi_\Delta}$, the negation $\boldsymbol{\neg}$ satisfies the condition $\boldsymbol{\neg} x = x \boldsymbol{\Rightarrow} 0$, and $\boldsymbol{\Delta}$ satisfies the condition $\boldsymbol{\Delta} x = x \boldsymbol{\eqcirc} 1$.

[1] The equality operator is crisp, i.e., the result is 0 or 1, while equivalence is not. By Guller's convention [17] we mostly use equality between atoms and constants as assertions, and equivalence between more complex formulae, but this is not a universal rule; see Ex. 19.

Definition 1 [12] Let *PropAtom* be the set of all propositional atoms. Let *OrdPropForm* be the set of all order propositional formulae constructed from *PropAtom*, the truth constants 0, 1, and the logical connectives of Π_Δ. An *order theory* is any subset of *OrdPropForm*.

Definition 2 [12] Let a mapping $\mathcal{V} : PropAtom \longrightarrow [0,1]$ be the valuation of propositional atoms such that $\mathcal{V}(0) = 0$ and $\mathcal{V}(1) = 1$. For any formula $\varphi \in OrdPropForm$, the value $\|\varphi\|^{\mathcal{V}} \in [0,1]$ of φ in $\mathcal{V}$ is defined recursively on the structure of φ:

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

Definition 3 [12] The formula $\varphi \in OrdPropForm$ has the *model* $\mathcal{V}$ (φ is true in the valuation $\mathcal{V}$, $\mathcal{V} \models \varphi$) iff $\|\varphi\|^{\mathcal{V}} = 1$. φ is *satisfiable* iff it has a model and is *valid* (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 formulae $\varphi \in T$. T is satisfiable iff it has a model and is valid iff every valuation is its model. For two formulae $\varphi, \varphi' \in OrdPropForm$, φ is equivalent to φ', or $\varphi \equiv \varphi'$ iff $\|\varphi\|^{\mathcal{V}} = \|\varphi'\|^{\mathcal{V}}$ for every valuation $\mathcal{V}$.

Order Clausal Form

The product fuzzy DPLL procedure introduced in section "Product DPLL Procedure" expects the input to be in *order clausal form*, which is the counterpart of normal forms in Boolean propositional logic. Below we revisit the definitions of the order clausal form and the associated notions to be able to refer to them later in the text.

Definition 4 [17] Let *power of atom* a^n be the n-th power of atom a interpreted by the $\cdot$ operator. Let *conjunction* Cn be a non-empty finite set of powers of atoms $\{a_1^{p_1}, \dots, a_n^{p_n}\}$ for $n \geq 1$ written as the expression $a_1^{p_1}$ & … & $a_n^{p_n}$. The atoms $a_i, 1 \leq i \leq n$ cannot occur more than once in the expression. An example of a conjunction is a^3 & b^4. Let *PropConj* designate the set of all conjunctions.

Definition 5 [17] Let *order literal* be an expression of the form $\varepsilon_1 \diamond \varepsilon_2$ where $\diamond \in \{\eqsim, \prec\}$ and $\varepsilon_i \in PropConj \cup \{0,1\}$. An order literal is called *equality literal* when $\diamond = \eqsim$, and *strict order literal* when $\diamond = \prec$. Two examples of order literals are a^2 & $b^3 \prec a$ and $a \eqsim 0$. An order literal is *pure* iff it does not contain any of the constants 0, 1.

Definition 6 [17] Let *order clause* be a set of order literals $\{l_1, \dots, l_n\}$ for $n \geq 1$ written as the expression $l_1 \vee \cdots \vee l_n$. Let $\square$ represent an *empty clause* $\emptyset$. A *unit clause* $\{l\}$ is a clause containing a single literal l. In contexts where this does not cause ambiguity, we write the unit clause $\{l\}$ as l, omitting the set braces. An example of an order clause is a^2 & $b^3 \prec a \vee a \eqsim 1$. An example of a unit clause is $a \prec 1$ (more formally $\{a \prec 1\}$).

Definition 7 [17] Let *order clausal theory* be a set of order clauses. An order clausal theory is {pure, unit} iff it contains only {pure, unit} clauses.

The interpolation rules used to perform the translation, along with a more comprehensive list of definitions, are to be found in Guller's work [17, Sect. 3].

Product DPLL Procedure

One of the well-established algorithms to solve the satisfiability problem in classical propositional logic is the DPLL procedure [10]. The algorithm performs backtracking to find a model of a given theory or prove its unsatisfiability.

The basic step consists of picking a literal and assigning to it the value of either *true* or *false* if possible. This step may be thought of as splitting the backtracking tree at the vertex representing a literal into two branches according to the assigned value. This branching step may be visualized in the form:

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

for literal l occurring in theory S. The product fuzzy extension of the DPLL procedure introduced by Guller [12] is also a backtracking-based algorithm that operates over finite order clausal theories and uses a similar kind of branching at its core. The branching step considers an atom and attempts to determine its value with the following trichotomy:

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

for atom a occurring in theory S.

Moreover, the classical DPLL algorithm employs two rules that constrain the search space—unit propagation and pure literal elimination. In product DPLL, there are seventeen rules (as defined in Guller's unpublished work), thirteen of which are necessary for the procedure to be refutation-complete, and four admissible rules that help constrain the search space or produce smaller trees.

The proof of satisfiability or validity of an order clausal theory is based on adequate application of the rules on the input theory. The rules split the tree into branches and simplify the clausal theory in the process. If a branch remains open after the non-deterministic application of all possible rules, there is a valuation of atoms under which the theory is true that may be obtained from the path of traversal.

The original proposal of the product fuzzy DPLL procedure may be found in the work of Guller [12] together with the related proof of refutational soundness and completeness.

Below we revisit some of the concepts and notation defined in [12] and Guller's unpublished work that we will refer to later in the paper.

Definition 8 [17] Let a be an atom and let the order clause C be a guard iff either $C = a \eqsim 0$, $C = 0 \prec a$, $C = a \prec 1$, or $C = a \eqsim 1$. Let S be an order propositional clause. Let $guards(a) = \{a \eqsim 0, 0 \prec a, a \prec 1, a \eqsim 1\}$ 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 $a \eqsim 0$, the literal $a \eqsim 1$, or both of the literals $0 \prec a, a \prec 1$.

Definition 9 Given $a \in PropAtom$, the propositional formula $a \eqsim 0 \vee 0 \prec a \wedge a \prec 1 \vee a \eqsim 1$ is a *trichotomy*. Given conjunctions $Cn_1, Cn_2 \in PropConj$, the pure order clause $Cn_1 \prec Cn_2 \vee Cn_1 \eqsim Cn_2 \vee Cn_2 \prec Cn_1$ is a *pure trichotomy*.

The DPLL rules make use of an auxiliary function and operation. First, the function $simplify : (\{0,1\} \cup PropConj \cup OrdPropLit \cup OrdPropCl) \times PropAtom \times \{0,1\} \to \{0,1\} \cup PropConj \cup OrdPropLit \cup OrdPropCl$ replaces every occurrence of a given atom in an input expression with the given truth constant according to laws holding in product algebra [17].

Definition 10 Auxiliary function *simplify* [17]

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

Example 1 Simplifying a clause
$simplify(a \prec 1 \vee a^2 \,\&\, b \eqsim 0, a, 1) = 1 \prec 1 \vee b \eqsim 0$

Next, $\odot : (\{0,1\} \cup PropConj) \times (\{0,1\} \cup PropConj) \to \{0,1\} \cup PropConj$ is a binary commutative and associative operator that returns the algebraic product of two conjunctions or literals according to Definition 11.

Definition 11 Auxiliary operation $\odot$ [17] Let Cn_1, Cn_2 be conjunctions and let the expression ε be a truth constant or a conjunction. Then the function $\odot$ is defined as

$$0 \odot \varepsilon = \varepsilon \odot 0 = 0;$$
$$1 \odot \varepsilon = \varepsilon \odot 1 = \varepsilon;$$
$$Cn_1 \odot Cn_2 = \{a^{m+n} \mid a^m \in Cn_1, a^n \in Cn_2\} \cup \{a^n \mid a^n \in Cn_1, a \notin atoms(Cn_2)\} \cup \{a^n \mid a^n \in Cn_2, a \notin atoms(Cn_1)\}$$

It can be extended to order literals $\odot : (\{0,1\} \cup OrdPropLit) \times (\{0,1\} \cup OrdPropLit) \to \{0,1\} \cup OrdPropLit$ in the following way: Let l_1, l_2 be order literals and the expression ε be a truth constant or an order literal.

$$0 \odot \varepsilon = \varepsilon \odot 0 = 0;$$
$$1 \odot \varepsilon = \varepsilon \odot 1 = \varepsilon;$$
$$l_1 \odot l_2 = (\varepsilon_1 \odot \varepsilon_2) \diamond (\upsilon_1 \odot \upsilon_2) \;\; if\; l_i = \varepsilon_i \diamond_i \upsilon_i,$$
$$\diamond = \begin{cases} \eqsim & if\; \diamond_1 = \diamond_2 = \eqsim, \\ \prec & else. \end{cases}$$

$\odot$ is a binary commutative and associative operator.

Example 2 Applying $\odot$ to conjunctions of powers and literals

$$a^2 \,\&\, b \odot b \,\&\, c = a^2 \,\&\, b^2 \,\&\, c$$
$$a^2 \eqsim b^3 \odot b \prec 1 = a^2 \,\&\, b \prec b^3$$
$$a \eqsim b \odot a \eqsim b \,\&\, c = a^2 \eqsim b^2 \,\&\, c$$

Next, we list and describe the intuition of the thirteen required fuzzy product DPLL rules as defined in Guller's unpublished work.

(*Unit contradiction rule*)

$$\frac{S}{S \cup \{\square\}};$$
$$\begin{array}{l} S \text{ is a unit order clausal theory;} \\ \text{there exist} \\ \quad 0 \prec a_0, \dots, 0 \prec a_m, \\ \quad a_0 \prec 1, \dots, a_m \prec 1 \in guards(S), \\ \quad l_0, \dots, l_n \in S \\ \text{such that } l_i \text{ is pure order literal and} \\ atoms(l_0, \dots, l_n) = \{a_0, \dots, a_m\}; \\ \text{there exist} \\ \\ \quad \alpha_i^* \geq 1, i = 0, \dots, n, \\ \quad J^* \subseteq \{j \mid j \leq m\}, \beta_j^* \geq 1, j \in J^*, \\ \text{such that} \\ \left(\odot_{i=0}^{n} l_i^{\alpha_i^*} \right) \odot \left(\odot_{j \in J^*} (a_j \prec 1)^{\beta_j^*} \right) \text{ is a contradiction.} \end{array} \tag{1}$$

If any $\odot$-product of powers of pure order literals or guards of the form $a \prec 1$ can be found that would lead to the contradiction of the form $\varepsilon \prec \varepsilon$, rule (1) derives $\square$ (closes the branch).

Example 3 Applying the unit contradiction rule

$$\frac{S = \{0 \prec a, 0 \prec b, a \prec 1, b \prec 1, a^2 \eqcirc b^3, b \prec a\}}{S \cup \{\square\}}$$

See the elaboration of this example in section "Unit Contradiction" Ex. 20.

(*Trichotomy branching rule*)

$$\frac{S}{S \cup \{a \eqcirc 0\} \mid S \cup \{0 \prec a, a \prec 1\} \mid S \cup \{a \eqcirc 1\}}; \quad a \in atoms(S). \tag{2}$$

The branching rule (2) splits the tree by assuming one of $a \eqcirc 0, 0 \prec a \prec 1, a \eqcirc 1$.

Example 4 Applying the trichotomy branching rule

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

(*Pure trichotomy branching rule*)

$$\frac{S}{(S - \{\varphi\}) \cup \{l_1\} \mid (S - \{\varphi\}) \cup \{C\} \cup \{l_2\} \mid (S - \{\varphi\}) \cup \{C\} \cup \{l_3\}};$$
$$\varphi = (l_1 \vee C) \in S, C \neq \square,$$
$$l_1 \vee l_2 \vee l_3 \text{ is a pure trichotomy.} \tag{3}$$

The branching rule (3) splits the tree into the three subcases of the trichotomy of pure literals l_1, l_2, and l_3.

Example 5 Applying the pure trichotomy branching rule

$$\frac{\{a \prec b \vee b \prec 1\}}{\{a \prec b\} \mid \{b \prec a, b \prec 1\} \mid \{a \eqcirc b, b \prec 1\}};$$

$a \prec b \vee b \prec a \vee a \eqcirc b$ is a pure trichotomy.

(*Contradiction rule*)

$$\frac{S}{(S - \{l \vee C\}) \cup \{C\}}; \quad l \vee C \in S, \ l \text{ is a contradiction.} \tag{4}$$

A contradictory literal is removed from a clause. Examples of contradictory literals are $1 \prec 0, 0 \eqcirc 1$.

Example 6 Applying the contradiction rule

$$\frac{\{1 \prec 0 \vee a \eqcirc b\}}{\{a \eqcirc b\}}$$

(*Tautology rule*)

$$\frac{S}{S - \{l \vee C\}}; \quad l \vee C \in S, \ l \text{ is a tautology.} \tag{5}$$

A tautologous literal is removed from S. Examples of tautologous literals are $0 \prec 1, 0 \eqcirc 0$.

Example 7 Applying the tautology rule

$$\frac{\{0 \eqcirc 0 \vee a \eqcirc b, a \prec b\}}{\{a \prec b\}}$$

(0-*simplification rule*)

$$\frac{S}{(S - \{C\}) \cup \{simplify(C, a, 0)\}}; \tag{6}$$
$$a \eqcirc 0 \in guards(S), \ C \in S, \ a \in atoms(C), \ a \eqcirc 0 \neq C.$$

If $a \eqcirc 0 \in guards(S)$ and the order clause C contains a, then C is simplified according to a and 0.

Example 8 Applying the 0-simplification rule

$$\frac{\{a \eqcirc 0, a \,\&\, b^2 \prec 1\}}{\{0 \prec 1\}}$$

(1-*simplification rule*)

$$\frac{S}{(S - \{C\}) \cup \{simplify(C, a, 1)\}}; \tag{7}$$
$$a \eqcirc 1 \in guards(S), \ C \in S, \ a \in atoms(C), \ a \eqcirc 1 \neq C.$$

Analogous to rule (6).

Example 9 Applying the 1-simplification rule

$$\frac{\{a \eqcirc 1, a \,\&\, b^2 \prec 1\}}{\{b^2 \prec 1\}}$$

(0-*contradiction rule*)

$$\frac{S}{(S - \{a_0^{\alpha_0} \,\&\, \cdots \,\&\, a_n^{\alpha_n} \eqcirc 0 \vee C\}) \cup \{C\}};$$
$$0 \prec a_0, \dots, 0 \prec a_n \in guards(S),$$
$$a_0^{\alpha_0} \,\&\, \cdots \,\&\, a_n^{\alpha_n} \eqcirc 0 \vee C \in S - guards(S). \tag{8}$$

If $0 \prec a_0, \dots, 0 \prec a_n \in guards(S)$, then obviously

$$a_0^{\alpha_0} \,\&\, \cdots \,\&\, a_n^{\alpha_n} \eqcirc 0$$

is contradictory and it is removed from the order clause.

Example 10 Applying the 0-contradiction rule

$$\frac{\{0 \prec a, 0 \prec b, a^2 \,\&\, b^3 \eqcirc 0 \vee c \prec 1\}}{\{0 \prec a, 0 \prec b, c \prec 1\}}$$

(1-*contradiction rule*)

$$\frac{S}{(S - \{a_0^{\alpha_0} \,\&\, \cdots \,\&\, a_n^{\alpha_n} \eqcirc 1 \vee C\}) \cup \{C\}};$$
$$a_i \prec 1 \in guards(S), i \leq n,$$
$$a_0^{\alpha_0} \,\&\, \cdots \,\&\, a_n^{\alpha_n} \eqcirc 1 \vee C \in S - guards(S). \tag{9}$$

Analogous to rule (8).

Example 11 Applying the 1-contradiction rule

$$\frac{\{a \prec 1, b \prec 1, a^2 \,\&\, b^3 \eqcirc 1 \vee c \prec 1\}}{\{a \prec 1, b \prec 1, c \prec 1\}}$$

(0-*consequence rule*)

$$\frac{S}{S - \{0 \prec a_0^{\alpha_0} \& \cdots \& a_n^{\alpha_n} \vee C\}}; \quad \begin{array}{l} 0 \prec a_0, \ldots, 0 \prec a_n \in guards(S), \\ 0 \prec a_0^{\alpha_0} \,\&\, \cdots \,\&\, a_n^{\alpha_n} \vee C \in S - guards(S). \end{array} \tag{10}$$

If $0 \prec a_0, \ldots, 0 \prec a_n \in guards(S)$, then obviously

$$0 \prec a_0^{\alpha_0} \,\&\, \cdots \,\&\, a_n^{\alpha_n}.$$

Therefore, the input order clause

$$0 \prec a_0^{\alpha_0} \,\&\, \cdots \,\&\, a_n^{\alpha_n} \vee C$$

is removed, as it is a consequence of the guard(s).

Example 12 Applying the 0-consequence rule

$$\frac{\{0 \prec a, 0 \prec b, 0 \prec a^2 \,\&\, b^3 \vee c \prec 1\}}{\{0 \prec a, 0 \prec b\}}$$

(1-*consequence rule*)

$$\frac{S}{S - \{a_0^{\alpha_0} \,\&\, \cdots \,\&\, a_n^{\alpha_n} \prec 1 \vee C\}}; \quad \begin{array}{l} a_i \prec 1 \in guards(S), \\ i \le n,\ a_0^{\alpha_0} \,\&\, \cdots \,\&\, a_n^{\alpha_n} \prec 1 \vee C \in S - guards(S). \end{array} \tag{11}$$

Analogous to rule (10).

Example 13 Applying the 1-consequence rule

$$\frac{\{a \prec 1, b \prec 1, a^2 \,\&\, b^3 \prec 1 \vee c \prec 1\}}{\{a \prec 1, b \prec 1\}}$$

(0-*annihilation rule*)

$$\frac{S}{S - \{a \eqcirc 0\}}; \quad a \eqcirc 0 \in guards(S),\ a \notin atoms(S - \{a \eqcirc 0\}). \tag{12}$$

(1-*annihilation rule*)

$$\frac{S}{S - \{a \eqcirc 1\}}; \quad a \eqcirc 1 \in guards(S),\ a \notin atoms(S - \{a \eqcirc 1\}). \tag{13}$$

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

Example 14 Applying the 0-annihilation rule

$$\frac{\{a \eqcirc 0, b \prec 1\}}{\{b \prec 1\}}$$

Example 15 Applying the 1-annihilation rule

$$\frac{\{a \eqcirc 1, b \prec 1\}}{\{b \prec 1\}}$$

Finally, we revisit three of Guller's admissible rules that help produce smaller branches.

(*Guard propagation rule I*)

$$\frac{S}{S \cup \{0 \prec b, b \prec 1\}}; \quad 0 \prec a,\ a \prec 1 \in guards(S),\ b \eqcirc a \in S. \tag{14}$$

The guardedness of an atom is propagated to other atoms bound by equality.

Example 16 Applying guard propagation rule I

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

(*Guard propagation rule II*)

$$\frac{S}{S \cup \{0 \prec c, c \prec 1\}}; \quad 0 \prec a,\ a \prec 1,\ 0 \prec b,\ b \prec 1 \in guards(S),\ c \eqcirc a \,\&\, b \in S. \tag{15}$$

If all of the atoms comprising a strong conjunction are fully guarded and are in equality with a single atom, the atom also becomes guarded.

Example 17 Applying guard propagation rule II

$$\frac{S = \{0 \prec a, a \prec 1, 0 \prec b, b \prec 1, c \eqcirc a \,\&\, b\}}{S \cup \{0 \prec c, c \prec 1\}}$$

(*Guard propagation rule III*)

$$\frac{S}{S \cup \{b \eqcirc 0\}}; \quad 0 \prec a, a \prec 1 \in guards(S),\ a \,\&\, b \eqcirc 0 \in S. \tag{16}$$

If a strong conjunction equals 0 and all atoms but one (b) are fully guarded, the equality $b \eqcirc 0$ is inferred.

Example 18 Applying guard propagation rule III

$$\frac{S = \{0 \prec a, a \prec 1, a \,\&\, b \eqcirc 0\}}{S \cup \{b \eqcirc 0\}}$$

In the next section, we provide a hypothetical real-world example that demonstrates a possible application of SAT solving in product propositional logic.

Motivational Example

Electricity suppliers face the risk of being unable to satisfy peak daily energy demand which often occurs in the morning and evening. This is especially true for the supply of energy from dispatchable sources, as solar power cannot be fully utilized during these times of day (the phenomenon is often described by the “duck curve”). To help balance demand, suppliers split the day into multiple zones where each zone is assigned a price per kilowatt hour. The simplest are splits into two zones (day and night rates), but some suppliers now provide programs wherein each hour of the following day has a predefined energy price based on predictions of demand.

Let us assume we wish to make use of such an hourly rate program when heating a well-insulated detached house with a heat pump. Let us also assume we have control over which hours the pump is active. To minimize costs, we wish to run the heat pump when the demand is low. However, the heat pump’s efficiency varies depending on the difference between the outside and inside temperature: the lower the difference, the more efficient the heat pump is. If we aim for a constant output temperature, then the efficiency is proportional to the temperature outdoors. Our goal is to have the heat pump active during the hours with low energy demand and high heating efficiency. Also, due to the insulation of the house, it is sufficient if the heat pump runs only two consecutive hours every six hours.

The problem of determining which hours to run the heat pump can be formalized as a fuzzy SAT instance. Our input data consist of bi-hourly prediction of energy demand (we have omitted odd hours for brevity) and the coefficient of performance (COP, unit-less) of the heat pump. Energy demand (in megawatts) is re-scaled into the interval [0, 1]. The COP cannot drop below 1 and is usually between 2 and 5 in European landlocked country spring days [20], so we will use the inverse of COP without further normalization. Since we seek to minimize both the normalized energy demand and the inverse COP, for simplicity, we will minimize their algebraic product. The data we use in this example are shown in Table 2.

To represent given data, our propositional product logic needs to be extended with the notion of intermediate constants (constants in the open interval (0, 1)) which this work does not yet cover. As these are useful for demonstrating the logic on a practical example, let us assume the support of such constants and their meaningful ordering; their inclusion is the subject of our ongoing work. Model finding is another current limitation of our work to be tackled in near future.

Now, we attempt to encode the problem. First, we need two sets of atoms to represent normalized energy demand (load) and inverse COP. The atoms $load_i$ and $icop_i$ will be set to the relevant values from Table 2 where i represents the hour of the day divided by two.

$$\begin{aligned} load_i &\eqcirc \langle \text{Load from table} \rangle_i \quad 0 \leq i < 12 \\ icop_i &\eqcirc \langle \text{iCOP from table} \rangle_i \quad 0 \leq i < 12 \end{aligned} \tag{17}$$

Table 2 Values of normalized energy demand (load), inverse coefficient of performance of the heat pump (iCOP), and their algebraic product (score)

Hour	Load	iCOP	Score
0	0.795	0.294	0.234
2	0.736	0.294	0.216
4	0.675	0.303	**0.205**
6	0.672	0.294	0.198
8	0.707	0.270	0.191
10	0.773	0.222	**0.172**
12	0.825	0.215	0.177
14	0.783	0.206	0.161
16	0.731	0.217	**0.159**
18	0.759	0.222	**0.169**
20	0.870	0.270	0.235
22	0.776	0.286	0.222

The values are fabricated, but were inspired by [25] and [20]

Next, we set the values $score_i$ to be equal to the strong conjunction of load and inverse COP (in other words, the score is load divided by COP).

$$score_i \eqcirc icop_i \,\&\, load_i \quad 0 \leq i < 12 \tag{18}$$

To fulfill the request that the heat pump ought to run two consecutive hours every six hours, we consider all consecutive six-hour partitions of scores to find the best split. As we work with bi-hourly data, the task is simplified into finding the single minimum score in every partition of consecutive scores of size three. One such partition is shown in Table 2; the column of scores is grouped into four subsets of size three as shown by the double row separator, and the minimum value is bold. Another such partition could be constructed by offsetting the groups by one, and the final partition by offsetting by two. For simplicity, we let the subsets wrap around in time, e.g. in the partition offset by two, the last group covers the hours 22, 0, and 2 (we do not reach into other days). This is represented by the atoms $minscore_{i,j}$ below (21), where i is the partition offset and j is the index of subset within the partition. The four bold scores in Table 2 are the minima $minscore_{0,\{0,1,2,3\}}$ found in the subsets of partition with offset 0. In our data, the displayed partition is also the “best” partition (one with the lowest minimum score)—in every other such split of the scores column, the minimum is higher.

Finally, we formulate the aforementioned condition in formulae (21, 22). The value of atom *max_of_best_part* captures the highest (worst) score in the best partition (one with the lowest minimum, 0.205 in Table 2).

$$minscore_{i,j} \eqcirc \bigwedge_{k=i+3j}^{(i+3(j+1)-1) \bmod 12} score_k \quad \begin{array}{l} 0 \leq i < 3, \\ 0 \leq j < 4 \end{array} \tag{20}$$

$$max\_of\_best\_part \asymp \bigwedge_{i=0}^{2} \bigvee_{j=0}^{3} minscore_{i,j} \tag{21}$$

$$0 \prec max\_of\_best\_part \tag{22}$$

The formulae (17–22) provide a solid foundation for our problem. The instance is constructed using the conjunction of all formulae:

$$\varphi_1 = (17) \wedge (18) \wedge (19) \wedge (20) \wedge (21) \wedge (22) \tag{23}$$

By performing satisfiability check of φ_1, our approach should yield a model containing valuations of the atoms. The fewest hours the heat pump would be running under the conditions in this example are shown in Table 2 where the score is bold, i.e., at least the hours 4–5, 10–11, 16–17, 18–19. These could be retrieved from the partition i where the maximum $minscore_{i,*}$ achieves the score of $max\_of\_best\_part$ (0.205 in the partition displayed by double row separators).

However, unless there is an entry with zero load, the formula φ is always satisfiable. To make the problem more interesting, we may impose constraints on the model, such as that the worst score of the best partition be less than a certain value. This constraint is formulated below within φ_2 and φ_3. Formula φ_2 is satisfiable in our data, but formula φ_3 is not.

$$\varphi_2 = \varphi_1 \wedge max\_of\_best\_part \prec 0.21 \tag{24}$$

$$\varphi_3 = \varphi_1 \wedge max\_of\_best\_part \prec 0.20 \tag{25}$$

In this example we have shown a way to utilize the product t-norm in the computation of score. Product propositional logic is useful to express relations where algebraic product or division can be suitably used. We could easily express the ratio of minimum vs. maximum score in our example by including the formulae

$$min \asymp \bigwedge_{i=0}^{11} score_i \tag{26}$$

$$ratio \leftrightarrow max \rightarrow min \tag{27}$$

Of course, product propositional logic alone has limited expressiveness. Suppose that we want to yield the models where the difference between $max\_of\_best\_part$ and min is more than 0.2, as these models may indicate periods of high energetic stress. While this is impossible in pure product logic, because difference requires Łukasiewicz negation ($\neg_Ł$ defined as $\neg_Ł\ x = 1 - x$) and equivalence ($\leftrightarrow_Ł$ defined as $x \leftrightarrow_Ł y = 1 - |x - y|$), it is possible to use these in Δ-extended product logic Π_Δ due to the embeddability of Łukasiewicz logic within Π_Δ [5]. The following formula might then be used after suitable embedding transformation:

$$range \leftrightarrow \neg_Ł(min \leftrightarrow_Ł max) \tag{28}$$

Obviously, a concrete real-world problem in the domain of heating, ventilation, and air conditioning (HVAC) would be much more complex—this example is purely demonstrational. Our purpose was to show a way fuzzy SAT solving may be used and how product propositional logic can be utilized. Other technologies could be used to solve the problem: SQL or its extensions, fuzzy answer set programming (FASP) [21], fuzzy description logics [1], a variant of mixed integer programming [7], or creating an ad hoc solution in a suitable programming language. The level of abstraction of the shown approach lies somewhere in the middle: on the one hand, it hides away the details of how the solver works and how auxiliary variables are introduced, so that the programmer may focus on specifying the problem. In this aspect it is a higher-level approach than writing a custom program or using mixed integer programming. On the other hand, this example may easily be deemed too complicated when compared to, e.g., FASP. Fuzzy answer set programming, however, may use a fuzzy SAT solver as its back-end, as FASP programs may be reduced to fuzzy SAT instances [29], which is one of the possible applications of our approach.

In the next section, we present the design of a deterministic algorithm that translates an input theory into order clausal theory and verifies its satisfiability or validity by the adequate application of the DPLL rules presented previously and performing the tree traversal.

Algorithm

In section "Product DPLL Procedure" we have outlined the intuition of the product fuzzy DPLL procedure and its tree-splitting and simplifying rules. In this section we define the algorithm that performs the translation of a theory into order clausal form, as well as the backtracking-based deterministic algorithm that performs the DPLL procedure to determine the satisfiability or validity of theories using the aforementioned rules.

The algorithm presented here is a revision of our previous work [28]. We have made several improvements which are clearly pointed out and discussed. We also revisit the parts of our work that have not been changed, for the convenience of the reader.

Translation into Order Clausal Form

As mentioned in section "Product DPLL Procedure", the DPLL procedure expects the input to be in order clausal form. The algorithm to translate an arbitrary theory in Π_Δ is based on the application of interpolation rules introduced by

```
    Input: Formula $\varphi$
    Result: Order clausal theory $S^{\varphi}$
 1  Function Translate($\varphi$):
 2      $assigned \leftarrow \emptyset$ (*);
 3      $queue \leftarrow [(\widetilde{a}_0, \varphi)]$;
 4      $S^{\varphi} \leftarrow \{\widetilde{a}_0 \eqcirc 1\}$ (SAT) or $\{\widetilde{a}_0 \prec 1\}$ (VAL);
 5      while $queue$ not empty do
 6          $\widetilde{a}_{\mathbb{i}}, \psi \leftarrow$ pick pair from $queue$;
 7          $nPairs, nClauses \leftarrow$ Inter($\widetilde{a}_{\mathbb{i}}, \psi$);
 8          foreach $pair \in nPairs$ do
 9              if subformula of $pair \in assigned$(*) then
10                  return associated auxiliary atom from
                      $assigned$(*);
11              end
12              add $pair$ to $queue$;
13              add $pair$ to $assigned$(*);
14          end
15          foreach $clause \in nClauses$ do
16              add $clause$ to $S^{\varphi}$;
17          end
18      end
19      return $S^{\varphi}$
```

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

Guller [17, Sect. 3], which may be thought of as a product fuzzy generalization of the translation of a Boolean formula into conjunctive normal form [24, 26].

Without loss of generality we consider the input to be a single propositional formula. If the intended input is a theory (a set of formulae), the procedure considers the input to be the $\wedge$-conjunction of its elements in Π_{Δ}.

The algorithm is shown in Alg. 1. Given a formula φ, the algorithm first generates the clause $\widetilde{a}_0 \eqcirc 1$ in the case of verifying satisfiability, or $\widetilde{a}_0 \prec 1$ in the case of verifying validity, where $\widetilde{a}_0$ is the auxiliary atom representing the full input formula. Then, the algorithm performs pre-order tree traversal over the structure of the formula. In every step, the intermediate formula ψ (either the initial formula or a subformula) is extracted from the queue along with the auxiliary atom $\widetilde{a}_{\mathbb{i}}$ associated with ψ. Then, ψ is processed by a compatible interpolation rule [17, Sect. 3]. For this purpose, assume the existence of function Inter$(\widetilde{a}_{\mathbb{i}}, \psi)$ that chooses the interpolation rule according to the connective of least precedence in ψ. The rule designates new order clauses $nClauses$ that are the result of the translation. Also, depending on the arity of the connective in ψ and the operands, the rule designates the subformula or subformulae ψ_i of ψ, $i \in \{0, 1\}$ that need to be translated further, and their corresponding auxiliary atoms $\widetilde{a}_{\mathbb{i}_i}$. These new subformulae and auxiliary atoms construct the list of pairs $nPairs$: $(\widetilde{a}_{\mathbb{i}_i}, \psi_i)$, $i \in \{0, 1\}$. The function Inter$(\widetilde{a}_{\mathbb{i}}, \psi)$ then returns $(nPairs, nClauses)$.

The improvement in the current revision as opposed to our previous work [28] is marked with (*). The problem of the previous algorithm can be illustrated on translating the formula $a \,\&\, b \rightarrow a \,\&\, b$. If the input formula contains multiple occurrences of identical subformulae, these have been treated as being separate, with one auxiliary atom associated with each. In the current version we employ the simple solution of remembering assigned subformulae and reusing the associated auxiliary atoms, effectively treating the formula as a directed acyclic graph rather than a tree. As a result, the search space of the DPLL procedure is now reduced by each repeated subformula.

Example 19 Translating a formula into order clausal form Let us consider the formula (27) from the motivational example in section "Motivational Example" modified by replacing $\leftrightarrow$ with $\eqcirc$ for demonstrational purposes:

$$\varphi = ratio \eqcirc (max \rightarrow min)$$

The translation into a SAT instance is depicted in the following steps, yielding the order clausal theory S^{φ}.

$$\{\widetilde{a}_0 \eqcirc 1, \widetilde{a}_0 \leftrightarrow \underbrace{ratio}_{\widetilde{a}_1} \eqcirc (\underbrace{max \rightarrow min}_{\widetilde{a}_2})\} \text{ [16, Tab. 1, Eq. 11]}$$

$$\{\widetilde{a}_0 \eqcirc 1, \widetilde{a}_1 \eqcirc \widetilde{a}_2 \vee \widetilde{a}_0 \eqcirc 0, \widetilde{a}_1 \prec \widetilde{a}_2 \vee \widetilde{a}_2 \prec \widetilde{a}_1 \vee \widetilde{a}_0 \eqcirc 1,$$
$$\widetilde{a}_1 \eqcirc ratio, \widetilde{a}_2 \leftrightarrow \underbrace{max}_{\widetilde{a}_3} \rightarrow \underbrace{min}_{\widetilde{a}_4}\} \text{ [16, Tab. 1, Eq. 9]}$$

$$\{\widetilde{a}_0 \eqcirc 1, \widetilde{a}_1 \eqcirc \widetilde{a}_2 \vee \widetilde{a}_0 \eqcirc 0, \widetilde{a}_1 \prec \widetilde{a}_2 \vee \widetilde{a}_2 \prec \widetilde{a}_1 \vee \widetilde{a}_0 \eqcirc 1,$$
$$\widetilde{a}_1 \eqcirc ratio, \widetilde{a}_3 \prec \widetilde{a}_4 \vee \widetilde{a}_3 \eqcirc \widetilde{a}_4 \vee \widetilde{a}_3 \,\&\, \widetilde{a}_2 \eqcirc \widetilde{a}_4,$$
$$\widetilde{a}_4 \prec \widetilde{a}_3 \vee \widetilde{a}_2 \eqcirc 1, \widetilde{a}_3 \eqcirc max, \widetilde{a}_4 \eqcirc min\}$$

$$\begin{aligned} S^{\varphi} = \{&\widetilde{a}_0 \eqcirc 1, \\ &\widetilde{a}_1 \eqcirc \widetilde{a}_2 \vee \widetilde{a}_0 \eqcirc 0, \\ &\widetilde{a}_1 \prec \widetilde{a}_2 \vee \widetilde{a}_2 \prec \widetilde{a}_1 \vee \widetilde{a}_0 \eqcirc 1, \\ &\widetilde{a}_1 \eqcirc ratio, \\ &\widetilde{a}_3 \prec \widetilde{a}_4 \vee \widetilde{a}_3 \eqcirc \widetilde{a}_4 \vee \widetilde{a}_3 \,\&\, \widetilde{a}_2 \eqcirc \widetilde{a}_4, \\ &\widetilde{a}_4 \prec \widetilde{a}_3 \vee \widetilde{a}_2 \eqcirc 1, \\ &\widetilde{a}_3 \eqcirc max, \\ &\widetilde{a}_4 \eqcirc min\} \end{aligned}$$

DPLL Inference

The algorithm performing the DPLL procedure accepts an order clausal theory as input and uses the rules (1)–(16) to split and simplify the theory tree. By the following definition 12 we obtain the answer to whether the theory is satisfiable.

Definition 12 [12] A branch is closed iff the empty clause $\square$ is derived, otherwise it is open. A tree is closed iff all its branches are closed, otherwise it is open. A theory is satisfiable iff an open branch exists (the theory has a model) once no more DPLL rules can be applied.

The flowchart in Fig. 1 provides an overview of the algorithm which is described in detail in Algs.2–6.

The parts of the algorithm described in this section have not been fundamentally changed since the previous version in [28]. However, we have revised the formulation of Alg.3 to be recursive, allowing for a simpler explanation of the trichotomy function. Also, the listing of Alg.4 has been updated to include Guller's admissible rules.

The algorithm begins with Alg. 2. If all atoms are fully guarded, skip trichotomy branching and reduce the theory using the `Reduce`(s) function. Otherwise, the non-empty input theory S is split by the function `Trichotomy`(S) at the first atom in the input theory that is not fully guarded, introducing the guards in branches according to the trichotomy branching rule (2). Every created branch s is then processed by the `Reduce`(s) function. If the branch cannot be recursively closed, it is open and the `Trichotomy`(S) function returns *true*. Otherwise all branches have been closed, S is unsatisfiable and the function returns *false*.

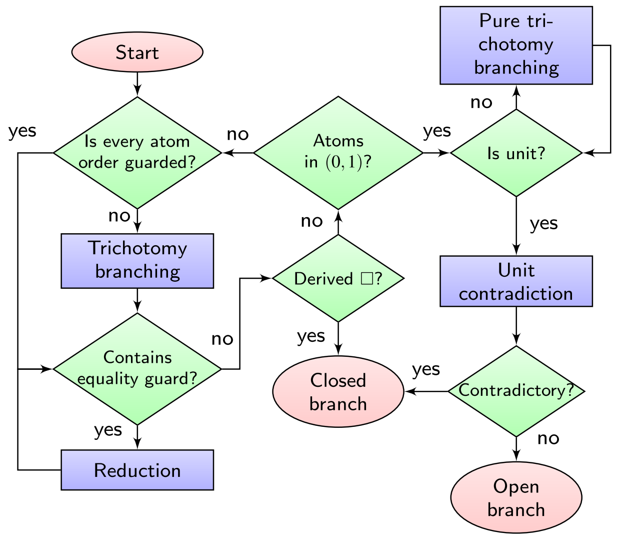

Fig. 1 Flowchart of the inference algorithm performing the DPLL procedure [28]

```
Input: A non-empty order clausal theory $S$
Result: $true$ iff $S$ is satisfiable
return Trichotomy($S$);
```

Algorithm 2. The initial step of the DPLL procedure [28]

```
Function Trichotomy($S$):
    if all atoms in $S$ are fully guarded then
        return Reduce($S$);
    end
    $a \leftarrow$ first atom in $atoms(S)$ that is not fully guarded;
    $S' \leftarrow$ split $S$ using rule (2) over $a$;
    foreach $s \in S'$ do
        if Reduce($s$) then
            return $true$;
        end
    end
    return $false$;
```

Algorithm 3. The `Trichotomy` function of the DPLL procedure [28]

The responsibility of the `Reduce`(S) function lies in the adequate application of the rules that reduce and simplify the theory. The rules are designed to eventually resolve all equality guards in the theory. Whenever a closed branch is derived, the function immediately returns *false*. Once all equality guards have been eliminated, we pass the computation to the `PureTrichotomy`(S) or `Trichotomy`(S) function according to whether all atoms in the theory are fully guarded or not, respectively. If any of the rules (5) or (10)–(13) manage to eliminate all clauses, the branch is considered open and the function immediately returns *true*.

The line marked with (*) indicates the change we have made in contrast to the previous version: in addition to the DPLL rules (4)–(13), we now also employ the admissible rules (14)–(16).

The function `PureTrichotomy`(S) described in Alg.5 is responsible for splitting the tree into branches according to the pure trichotomy rule (3). First, any equality guards introduced by the application of the trichotomy rule are handled

```
Function Reduce($S$):
    while $S$ contains equality guard do
        $S \leftarrow$ Application of rules (4)–(16)(*) consecutively.
          Return $false$ if any of the rules returns $false$.
          Return $true$ if any of the rules (5), (10)–(13) return
          $true$;
    end
    if all atoms in $S$ are fully guarded then
        return PureTrichotomy($S$);
    end
    return Trichotomy($S$);
```

Algorithm 4. The `Reduce` function of the DPLL procedure [28]

by the `Reduce`(S) function. Then, the special case of S being a unit theory is handled by the `UnitContradiction`(S) function. Otherwise the algorithm attempts to process every non-unit clause with the intention to split it into at most three branches. The literal a of clause Cl is first checked for occurrences in other clauses. In the case the literal occurs in an existing unit clause (it is already assumed to be *true*), remove all clauses containing the literal and continue with this assumption. Otherwise its counterpart literals b and c that form the pure trichotomy $a \vee b \vee c$ are generated[2]. If any of these literals is already assumed to be *true* in the other clauses, a cannot be true, so we remove every occurrence of a from the theory. Otherwise (if neither of these literals occurs in any unit clause of the theory), we split the theory at line 18 according to rule (3) and remove the occurrences of the complementary literals (assumed to be *false*) in the respective branches. If a closed branch is derived in any of the branches, return *false*, otherwise continue the traversal recursively over each created branch.

```
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 $Cl \in S'$ do
6          $a \leftarrow$ first literal in $Cl$;
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 the 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 return $true$;
22         end
23         return $false$;
24     end
25     return $false$;
```

Algorithm 5. The `PureTrichotomy` function of the DPLL procedure [28]

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

```
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 theDPLL procedure [28]

Finally, the function `UnitContradiction`(S) shown in Alg.6 attempts to apply the unit contradiction rule (1). The function returns *true* iff there exists a $\odot$-product of powers of literals appearing in the theory that is contradictory (therefore the branch is closed).

Limitations

If the branch remains open after the application of the unit contradiction rule, the theory is satisfiable has a model. However, the current state of the solver does not perform model-finding.

It is also important to note that in this stage, the algorithm does not have support for intermediate constants (constants other than 0, 1) in the input theory.

Unit Contradiction

As described in section "Product DPLL Procedure", the unit contradiction rule (1) involves the problem of finding the contradiction of the form $\varepsilon \prec \varepsilon$ if there is any possibility to yield it using the operation $\odot$ over pure order literals and the strict order guards $a_i \prec 1$ present in the theory. More simply, it is the problem of selecting the powers of literals in order to yield such a contradiction using their $\odot$-product.

Example 20 from our previous work [28] illustrates this problem on the theory containing the literals $\{\mathrm{a}^2 \eqcirc \mathrm{b}^3, \mathrm{b} \prec \mathrm{a}\}$ and the guards of atoms a and b $\{0 \prec a, 0 \prec b, a \prec 1, b \prec 1\}$. In this example we can form a contradiction by using the boxed literals: literal (30) with the power of 2 and literals (29, 31) as they are. By performing the operation $\odot$ over these powers of literals, we yield the contradiction $a^2\&b^3 \prec a^2\&b^3$ (32).

Example 20 Application of the unit contradiction rule [27]

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

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

$$a^2 \eqcirc b^3 \tag{29}$$

$$(b \prec a)^{\mathbf{2}} \tag{30}$$

$$b \prec 1 \tag{31}$$

$$a^2 \,\&\, b^3 \prec a^2 \,\&\, b^3\text{—a contradiction} \tag{32}$$

Remark 1 Note that according to Definition 11, at least one of the literals used in the product has to be a strict order literal in order to yield the contradiction $\varepsilon \prec \varepsilon$. Otherwise the resulting literal would be an equality literal.

Our previous paper [28] introduces a method to solve this problem by using linear programming (LP). The canonical form of a linear program is:

$$\begin{array}{ll} \text{find a vector} & x \\ \text{that minimizes} & c^T x \\ \text{subject to} & Ax \diamond b \\ \text{and} & x \geq 0 \end{array}$$

where A is the matrix of coefficients of variables x to be determined, b the vector of constraints, c the vector of objective function coefficients, and $\diamond$ the vector of order relations.

We define encoding of the input theory into an LP instance in Definition 13 as follows:

Definition 13 Given a unit order clausal theory, let m be the number of atoms, q the number of equality literals, and p the number of strict order literals of the theory. The LP encoding of the unit contradiction problem is then

$$A = \begin{bmatrix} a_{1,1} & \dots & a_{1,2q} & a_{1,2q+1} & \dots & a_{1,2q+p} \\ \vdots & \ddots & \vdots & a_{1,2q+1} & \ddots & \vdots \\ a_{m,1} & \dots & a_{m,2q} & a_{1,2q+1} & \dots & a_{m,2q+p} \\ 0 & \dots & 0 & 1 & \dots & 1 \end{bmatrix}$$

$$[1ex]b = [\overbrace{0, \dots, 0}^{m}, 1]^T$$

$$[1ex]c = [\overbrace{1, \dots, 1}^{2q+p}]^T$$

$$[1ex]\diamond = [\overbrace{=, \dots, =}^{m}, \geq]^T$$

The variables x of the LP problem to be determined represent the powers of pure order literals and the guards $a \prec 1$ of the theory, and the constraints (the rows in matrix A) represent the atoms. The equality literals in the matrix A are represented twice (once with the operands reversed) to handle the commutativity of the equality operator $\eqsim$. The coefficients of the variables are set to the difference of powers of atoms appearing on the left- and right-side of literals[3].

[3] E.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 $0 - 1 = -1$ [28].

To constrain the variables so that at least one order literal is used as per remark 1, the constraint coefficients in the last row of A, the last element of b, and the last relation in $\diamond$ are set accordingly. The objective of such LP is *minimize*.

Remark 2 In our previous work [28] we have defined a similar encoding. However, when adding the constraint to reject invalid cases (the last row of matrix A), we did not take into account the rejection of cases where no order literals were used to form the contradiction—we only considered the cases in which no literals were used whatsoever.

The coefficient matrix for the theory in Ex. 20 according to Definition 13 is shown in Eq. (33).

$$A_{ex} = \begin{bmatrix} 2 & -2 & -1 & 1 & 0 \\ -3 & 3 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix} \tag{33}$$

Remark 3 In this paper we highlight the advantage of our solver in not relying on translations into other solvers, but we do make use of LP to solve the unit contradiction problem. However, the use of an external LP solver is isolated to solving this problem only, which is not necessarily invoked while solving the satisfiability or validity of formulae. Nevertheless, we are considering replacing even this step with a custom solution.

Implementation

In our previous work [28] we have introduced and described the first working implementation of the solver and named it prodfsat. In this section we describe its current version, although the interface and technical details contain only minor changes. The implementation is available for download[4] and is free to use under the GNU General Public License v3.0 or later.

The software consists of several executable binary artifacts:

- **prodfsat** is the main console application that parses an input product propositional theory and outputs the results of SAT or VAL solving,
- **prodfsat_tests** executes the defined test suites, which include unit tests as well as the set of example theories used to conduct experiments,
- **prodfsat_niblos_converter** converts the input formulae into representation for the NiBLoS [31] and mNiBLoS [30] solvers for the use in experiments,

[4] https://git.uhliarik.com/ivor/prodfsat

```
prodfsat [-s] [-p] [-t sat|val] [FILE]...
    [-- THEORY...]
```

Generation of random formulae with given length and number of atoms

Table 3 The mapping of text strings to logical expressions

Connective	Text string	Example
power	`“^”`	`“a^3”`
$\neg$	`“-”`	`“-a”`
&	`“&”`	`“a & 0 & b^3”`
$\eqcirc$	`“=”`	`“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”`

- **prodfsat_niblos_random** generates random formulae used in experiments,
- **prodfsat_niblos_hard** generates instances of hard-to-solve formulae used in experiments.

In the remainder of this section we provide an introduction to using the main application, demonstrate running it on an example, and briefly describe technical details.

Application Usage

The main application may be run in console according to the following specification:

where

- the switches `-s`, `-p` enable the debug messages of scanning and parsing,
- the switch `-t` controls whether SAT-solving or VAL-solving is to be performed (the default mode is `sat`),
- the list of positional arguments `FILE...` are the files containing the input product propositional theories,
- alternatively, the input may be passed as string in the list of positional arguments following `-`.

The syntax of formulae accepted by the program is the same as in the previous version [28] and follows the mapping listed in Tab. 3. The operator precedence matches that of $\mathbf{\Pi}_\Delta$ and can be overridden with parentheses. The names of atoms must begin with an alphabetic character and follow with any number of alphanumeric characters. The constants representing falsehood and truth $(0, 1)$ are `0` and `1`, respectively. The powers of atoms must be non-zero positive integers. The lines starting with the character # are considered comments and are skipped by the program. The input may contain multiple theories separated by two or more consecutive newline characters. The Δ connective is currently not supported, but the semantically equivalent expression $x \eqcirc 1$ may be used to represent Δx.

As stated previously in section “Translation into Order Clausal Form”, a theory is interpreted as the $\wedge$-conjunction of the theory’s formulae. To facilitate this, formulae joined by a comma or a single newline character are parsed as a single theory, but in this case as conjunction with the lowest operator precedence, as is listed in the last row of Table 3.

Remark 4 As the current version of prodfsat does not support model-finding in the traditional sense of the interpretation of atoms, the mention of a model in the source code of the project refers to the set of literals that are true in an open branch.

Example

Here we demonstrate executing the main program to prove the validity of the formula in Eq. (34) (one of basic logic axioms [19]).

$$(\varphi \,\&\, \psi) \rightarrow (\psi \,\&\, \varphi) \tag{34}$$

We first encode the formula into the following plain-text form:

```
(a & b) -> (b & a)
```

Then, we run the program with the adequate switch and the formula as its positional argument:

```
prodfsat -t val -- "(a & b) -> (b & a)"
```

The program parses the input and performs translation into order clausal form for solving VAL (where each auxiliary atom is marked with an asterisk and its index). Then, the program outputs whether the formula is valid.[5]

[5] The program output has been modified for better readability.

```
[info] Processing result:
((a & b) -> (b & a))
[info] VAL: In clausal form:
(((*0) < (1)))
(((*1) < (*2)) V ((*1) = (*2))
    V ((*1 & *0) = (*2)))
(((*2) < (*1)) V ((*0) = (1)))
(((*1) = (a & b)))
(((*2) = (b & a)))
[info] VAL: true
```

Technical Details and Improvements

As in the previous version [28], the application is implemented in the C++ language, leveraging some of the features of modern standards up to C++20. The data structures in the source code were designed to maintain intuitive representation, but keeping performance measures in mind (efficient memory handling, utilizing move semantics, allowing for copy elision to take place when possible, etc.).

Apart from the enhancements of the algorithm, we have made changes to this version regarding the memory layout of objects representing clausal formulae to achieve better cache locality, as well as employed other optimizations. The most notable of these is the truncation of the tree resulting from translation into order clausal form in the case of strong conjunctions of powers: if a subformula is a strong conjunction of powers with only atoms and constants, e.g., a^3 & b & c^2, we do not break this further down. Instead, the formula remains in the leaf and can be processed by the DPLL procedure directly. Also, the code has been refactored with the aim to make it easier to extend the DPLL procedure with additional reduction rules. In addition, we have implemented the admissible DPLL rules (14)–(16).

The implementation has been tested in the Linux environment, but should be operating-system agnostic. The project's website contains the complete list of dependencies, as well as instructions on how to build the project and run the main program.

Experimental Results

To verify the implementation of prodfsat we have conducted a number of experiments. In each we measured the runtime of the program and where possible compared it with the runtime of other existing solutions, namely the mNiBLoS solver [30], its predecessor NiBLoS [31], and the previous version of prodfsat [28]. Although the mNiBLoS solver is more advanced than NiBLoS in the case of the product t-norm (especially due to utilizing the isomorphism between the standard product algebra and $\mathbb{R}^-_{\bullet}$ [30, Sect. 3.1.2] which avoids algebraic multiplication), we include it in a part of our comparative test bench for wider reference.

The decision to use mNiBLoS and NiBLoS in our comparisons was made due to practical reasons: they intersect with prodfsat in terms of being able to solve the SAT and VAL problems over product propositional logic, they are not based on stochastic methods, and their implementations were readily available and adaptable. However, it is important to note the crucial differences: (1) both of these two projects have more general domains than prodfsat, as they operate over the ordinal sums of the three fundamental t-norms; (2) mNiBLoS supports not only propositional logic but also the modal expansion; (3) these projects have support for intermediate rational constants which prodfsat currently lacks; (4) they are based on the translation into SMT problem instances and rely on an SMT solver, while the core part of prodfsat is self-contained. Due to the different goals of our work and [m]NiBLoS, we do not consider these projects competitive, although we still find the comparison important to derive conclusions about the state of our work.

The experiments consist of five parts. First, we measure the runtime using the set of test inputs from our previous work [28] that are processed in batch. Then we compare the performance of prodfsat, mNiBLoS, and NiBLoS over the conforming subset of these inputs one-by-one. Afterward, inspired by the experiments by Vidal [30], we compare the implementations over a fixed formula with varying powers of atoms, randomly generated formulae, and a hard problem consisting of formulae with a high number of atoms.

Methodology

To conduct comparative experiments we have adapted the source code of mNiBLoS to support non-interactive input. Next, we have developed a program that converts the syntactical representation of formulae specific to prodfsat to both NiBLoS and mNiBLoS. Due to differences in acceptable input between these three systems, we have either limited the generation of formulae to the common subset of expressions (in the case of random formulae in section "Randomly Generated Formulae") or excluded the test examples that contained connectives unsupported by existing solutions (in the case of the test set in sections "Test Set (Batch)" and "Test Set (One-by-one)").

In the internal evaluation of prodfsat and the comparison of the current version with our previous work [28] in section "Test Set (Batch)", the measurements cover only the time needed to read and parse the input data and perform satisfiability and validity proofs, i.e., the time

Table 4 Runtime in milliseconds over test examples processed in batch using our previous work and the current version of prodfsat

Group	Size	Old version			Current version		
		Mean	SD	NM	Mean	SD	NM
BLAxioms	8	709.2	43.51	88.65	142.8	3.65	17.85
BLConj	8	966.4	35.75	120.80	292.5	3.89	36.56
BLConstant	3	4.4	0.55	1.47	2.5	0.53	0.83
BLDisj	8	953.2	12.32	119.15	289.5	4.12	36.19
BLEquiv	9	1017.2	15.56	113.02	298.7	7.57	33.19
BLImpl	3	107.0	2.55	35.67	27.8	2.74	9.27
BLNegation	6	54.6	3.44	9.10	19.7	1.06	3.28
BLStrongConj	6	706.2	12.83	117.70	161.3	6.55	26.88
BLMisc	6	519.2	25.17	86.53	94.6	3.63	15.77
ProductLogic	5	115.4	2.51	23.08	33.0	1.56	6.60
Guller	2	234.6	4.56	117.30	53.4	1.65	26.7
Custom	5	135.2	6.14	27.04	87.4	5.50	17.48
Total	**69**	**5587.4**	**90.03**	**80.98**	**1503.2**	**26.28**	**21.79**

SD is the corrected sample standard deviation, NM is the mean value normalized by the number of tests in the group. Values across all test examples are shown in bold

required by the operating system to load the program is omitted. More specifically, the tests are performed and timed using the Google Test framework. However, as the [m]NiBLoS systems employ a completely different approach and are implemented in a different programming language, we have chosen to measure the entire time of running the process from the command line with the related experiments[6].

Remark 5 We have observed that—at least in the case of non-trivial inputs—the translation portion of runtime of either NiBLoS or mNiBLoS is negligible in comparison to its execution of the Z3 SMT theorem prover. We have considered measuring the time of the Z3 prover alone; however, as our intention was to include the time of prodfsat's translation into order clausal form in the total runtime, we have decided to stay symmetric with this decision.

To avoid inconsistencies, each measurement in this work has been performed 10 times with the same input unless otherwise stated. All experiments have been conducted on the same hardware[7].

There have been instances of problems where the computation timed out according to the threshold set by the test in question. As the results are averaged over multiple runs with the same input, we treat timeouts with the following dichotomy: if the computation timed out at least half of the times (usually at least 5 times), we declare the average value as timed out; otherwise the timed-out runs are excluded from the average.

Test Set (Batch)

In our previous work [28] we measured the performance of the earlier version of prodfsat using the set of 69 test formulae. These examples are mostly composed of formulae from literature: axioms of Hájek's basic logic (8 formulae), properties of basic logic (49 formulae), product logic axioms and properties (5 formulae) [19]; examples by Guller (2 formulae); custom examples created during the development of prodfsat (5 formulae)[8]. As the execution over some of the examples is too fast to adequately measure the runtime in milliseconds, we have joined them into 12 groups based on their occurrence in literature (e.g., "BLAxioms" are the axioms of basic logic, "BLMisc" are six properties of basic logic, "Custom" are the five examples created in our work).

We have performed the tests using the program `prodfsat_tests`[9] to measure the performance of the current version of prodfsat with the algorithm and implementation enhancements described in previous sections. Both satisfiability and validity proofs have been performed for every

[6] The shell scripts used to perform the experiments are part of the project's files.

[7] The experiments have been conducted on a personal computer with the CPU frequency of 3.31 GHz; the current implementation of prodfsat is single-threaded.

[8] The full list of formulae may be found in tests that are part of the project's files and are used by the `prodfsat_tests` program in the test suite `Solving`.

[9] Built with GCC 11.1.0, optimization level 3, and link-time optimization.

Table 5 Runtime in milliseconds over test examples processed one-by-one for the VAL problem

Group	Size	prodfsat	mNiBLoS	NiBLoS
BLAxioms	8	**39.02**	61.89	85.32
BLConj	8	58.31	58.46	**54.65**
BLConstant	3	**20.29**	49.42	43.39
BLDisj	8	58.34	60.04	**50.64**
BLImpl	3	**30.01**	54.53	51.84
BLNegation	6	**23.69**	60.42	113.59
BLStrongConj	5	**34.23**	71.29	82.93
BLMisc	6	**39.86**	57.75	54.63
ProductLogic	5	**26.48**	58.20	130.56
Total	**52**	**330.22**	**532.00**	**667.55**

The values are normalized by the number of tests in the group. Minima across solvers are highlighted in bold

Table 6 Runtime in milliseconds over test examples processed one-by-one for the SAT problem

Group	Size	prodfsat	mNiBLoS	NiBLoS
BLAxioms	8	**20.43**	60.35	58.35
BLConj	8	**19.68**	54.78	54.18
BLConstant	3	**18.94**	50.73	44.32
BLDisj	8	**19.48**	56.98	51.04
BLImpl	3	**19.33**	54.02	54.69
BLNegation	6	**20.07**	61.85	60.52
BLStrongConj	5	**20.49**	61.12	59.79
BLMisc	6	**20.07**	56.71	54.63
ProductLogic	5	**19.90**	62.89	62.59
Total	**52**	**178.39**	**519.43**	**496.86**

The values are normalized by the number of tests in the group. Minima across solvers are highlighted in bold

example. The resulting timings in milliseconds are the averages of 10 measurements (the values for the old version were taken from our previous paper [28] where 5 measurements were used), with the system's cache cleared between each run, and are shown in Table 4.

Upon inspection, it is clear the enhancements are substantial when proving SAT and VAL of these formulae, with the total average runtime reduced down to 1503.2 ms, 26.9% of the previous version of prodfsat. The average runtime divided by the number of tests is therefore 21.79 ms. Several individual tests ($0 \rightarrow \phi$, $\phi \eqcirc \phi$, etc.) reported the runtime rounded down to 0 ms, assuming the SAT and VAL solving together took only a few hundred microseconds. In these cases, the algorithm applies the reduction rules and eliminates all clauses (SAT) or finds a contradictory clause (VAL) in only a few steps.

As the formulae used in this experiment are relatively short in the number of connectives and atoms, the low values in the execution times are expected. Nevertheless, the results confirm the enhancements made in this work. The comparison with other solvers as well as solving more complex examples follow in the next experiments.

Test Set (One-by-one)

In this experiment we compare the performance of prodfsat with mNiBLoS and NiBLoS over the examples from section "Test Set (Batch)". We have excluded the tests with connectives unsupported[10] by either NiBLoS or mNiBLoS (equivalence and the strict order between an atom and one of 0, 1). The main difference from the previous experiment is the nature of obtaining the timings: as per section "Methodology", this and all the following experiments in this work have been conducted by measuring the time of the execution of the whole program on each test example one-by-one. Therefore, even the results for prodfsat are higher in duration than in section "Test Set (Batch)". Moreover, while in the previous experiment we have measured the total duration of proving both satisfiability and validity, we list the two separately throughout the rest of the paper for the sake of consistency with Vidal [30].

The results can be seen in Table 5 for proving validity and Table 6 for satisfiability. In most cases of the former, prodfsat finishes faster, although sometimes only marginally. On the other hand, prodfsat finishes faster when proving the satisfiability of formulae. The faster execution time of SAT-solving as opposed to VAL may be explained by the nature of the test examples. The majority are valid (axioms, properties), therefore the algorithm of prodfsat has to traverse every branch of the fuzzy DPLL tree, whereas to prove satisfiability, the algorithm stops at the first closed branch. Nevertheless, similarly to the previous experiment, the results show that prodfsat performs reasonably well over short formulae.

Parameterized Power

In the evaluation of mNiBLoS, Vidal has performed a comparative analysis using the test bench of generalizations of axioms of basic logic [30, Sect. 4.2] with the varying parameter n. One of these generalizations is shown in the parameterized formula in Eq. (35).

[10] According to the associated documentation and our belief.

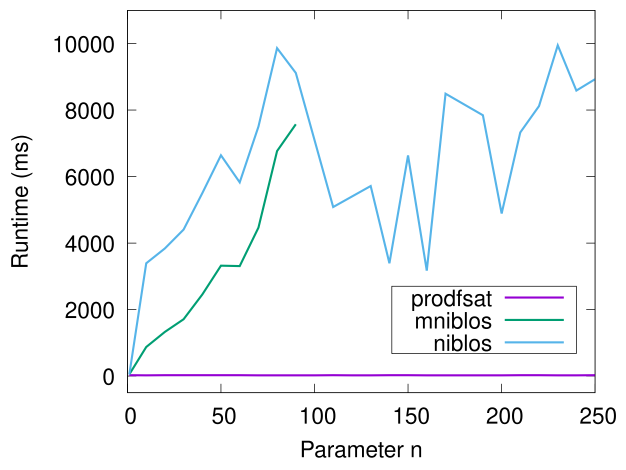

Fig. 2 Runtime in milliseconds required to prove the validity of the formula $(\varphi^n \& \psi^n) \to (\psi^n \& \varphi^n)$ with increasing n in steps of 10. Values exceeding the timeout threshold of 10 seconds are not shown

$$(\varphi^n \& \psi^n) \to (\psi^n \& \varphi^n) \tag{35}$$

Vidal's work shows that in the case of product logic, the runtime of mNiBLoS needed to prove validity increases polynomially with increasing n. To display one of the advantages of prodfsat, we have reconstructed the experiment (for each n with the increment of 10 the measurement was taken only once) and report the results in Fig. 2. The runtime values and the polynomial complexity of proving validity w.r.t. the parameter n by mNiBLoS coincide with the measurements of Vidal. The constant complexity of prodfsat (with the average duration of 23.52 ms) is given by the direct representation and processing of powers of atoms, which are safely eluded in this case.

Randomly Generated Formulae

The paper introducing mNiBLoS proposes an interesting experiment to test the solver on more irregular examples [30, Sect. 4.2]. This is done by the random generation of formulae of varying length, in terms of the connectives and the number of atoms, and the varying number of atoms used in these formulae. In addition, the generation was performed in two modes: with and without constants. Inspired by the design, we have decided to recreate the experiment for both prodfsat and mNiBLoS. The precise way of generating random formulae in the paper [30] is not known to us, therefore we present our algorithm in the pseudocode 7.

The pseudocode in Alg.7 shows our bottom-up construction of random formulae, where the leaves are atoms or constants, and inner vertices are connectives. There are always at least *atomCount* unique atoms generated. Because the tree

```
Result: Generated formula tree
1  Function Produce(length, atomCount):
2      subtrees ← GenerateLeaves(length, atomCount);
3      while size(subtrees) > 1 do
4          left, right ← extract two elements from subtrees;
5          newLeaf ← join left and right with random
             connective;
6          add newLeaf to subtrees;
7      end
8      return subtrees[0]

Result: List of leaves
9  Function GenerateLeaves(length, atomCount):
10     leaves ← create atomCount unique atoms;
11     foreach 0 < i < length/2 + 1 − atomCount do
12         if generate constants then
13             newLeaf ← generate random atom or constant
                 at random;
14         end
15         else
16             newLeaf ← generate random atom;
17         end
18         add newLeaf to leaves;
19     end
20     return leaves
```

Algorithm 7. Generation of random formulae with given length and number of atoms

is binary, and *length* is the total size of the tree, there must be $length/2 + 1$ leaves. Therefore, once *atomCount* atoms have been generated, the rest is filled with reoccurring atoms or with constants at random when enabled.

To ensure compatibility between prodfsat and mNiBLoS, the connectives are limited to conjunction, disjunction, implication, and strong conjunction. All atoms are generated with the power of 1, and constants are limited by prodfsat to 0 and 1.

We have executed several sets of tests over randomly generated formulae with varying length and number of atoms with all three solvers. For brevity, we omit the results of NiBLoS in these tests. The runtime measured in milliseconds is shown as heat-maps in Figs. 3, 4, 5, 6 in decimal-logarithmic scale. Timeouts with the threshold of 1 min are represented with white gaps (below the diagonal). At least one atom was generated in all tests. Moreover, each test was split between two intervals of varying length: 3–47 with increments of 4 and atom count increments of 2, and 51–291 with increments of 20 and atom count increments of 10.

First, we have tested the performance of proving the validity of random formulae without constants. The results for prodfsat and mNiBLoS are shown in Figs. 3 (length up to 47) and 4 (length up to 291). The difference between the two solvers is immediate: while prodfsat performs consistently better over short formulae (length up to 15), it struggles with

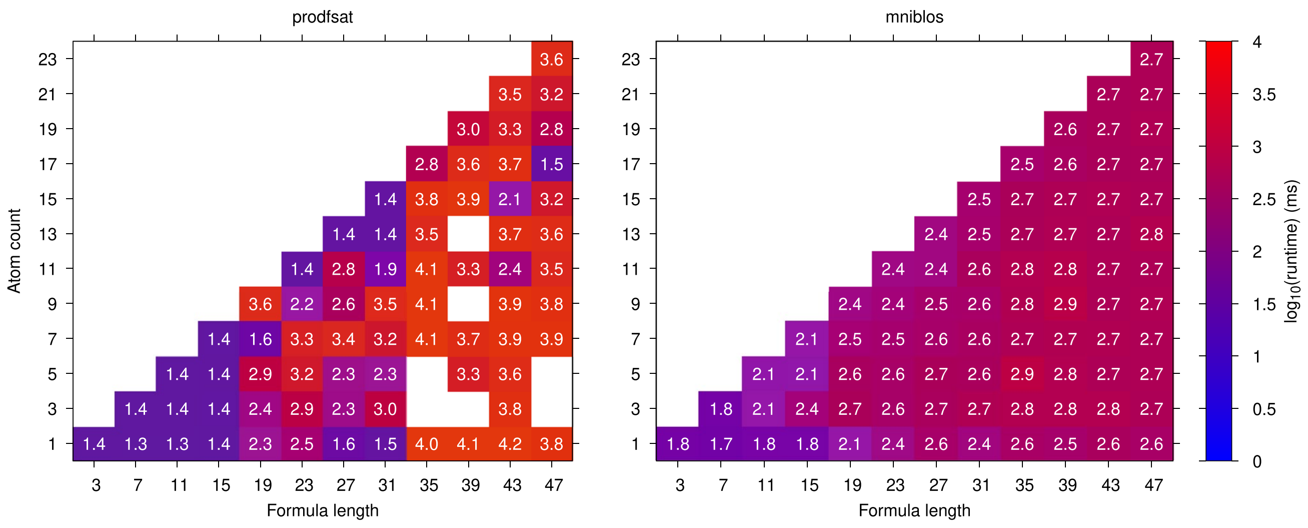

Fig. 3 Runtime of prodfsat (left) and mNiBLoS (right) in decimal-logarithmic scale required to prove the validity of random formulae with increasing length (3–47) and number of atoms. Values exceeding the timeout threshold of 1 min are not shown

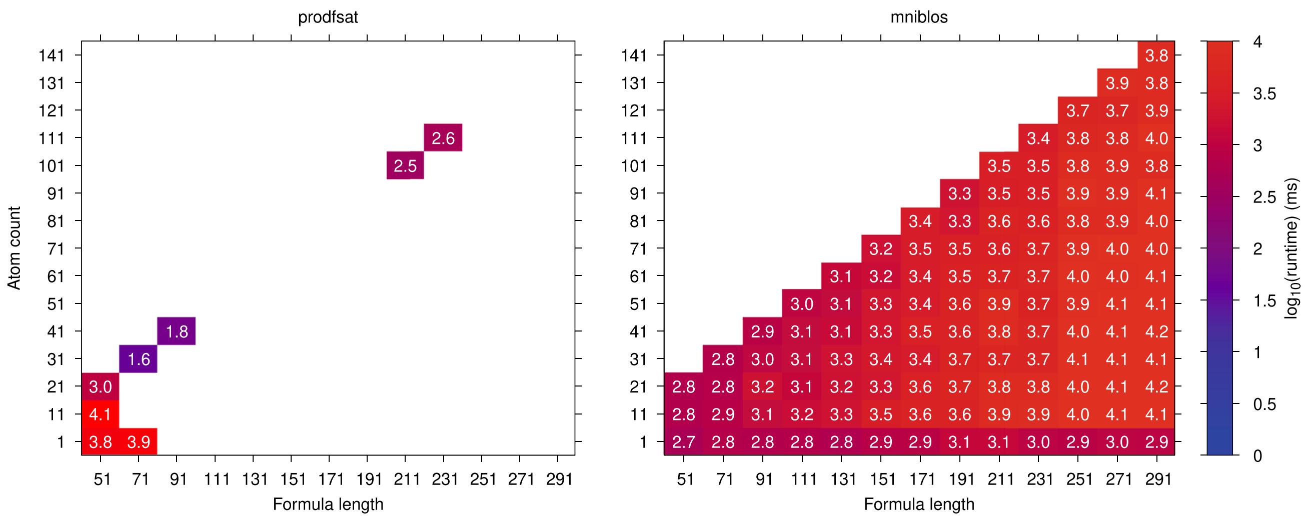

Fig. 4 Runtime of prodfsat (left) and mNiBLoS (right) in decimal-logarithmic scale required to prove the validity of random formulae with increasing length (51–291) and number of atoms. Values exceeding the timeout threshold of 1 min are not shown

higher values, with 1-min timeouts occurring at length 35. In contrast, mNiBLoS is efficient at proving the validity of all tested formulae. Interestingly, Fig. 4 shows that prodfsat can sporadically perform well (in some cases an order of magnitude better than mNiBLoS). One such case occurs with length 231 and 111 unique atoms. The average measured time of prodfsat was 426.18 ms, while it took mNiBLoS 2748.47 ms. We hypothesize this is due to the unpredictable applicability of fuzzy DPLL reduction rules to random formulae.

The situation is slightly better with introduced constants as can be seen in Fig. 5. With an increasing number of constants, the repetitions of atoms are decreased (but all still occur at least once), and they may be thought of as constraints in the solution space for both prodfsat and mNiBLoS. As a result, prodfsat outperforms mNiBLoS up to the length of 30. With greater lengths, however, the results become similar to those over formulae without constants.

The results of proving satisfiability without constants, which are shown in Fig. 6, are again similar to that of proving validity, with even more occurrences of timeouts in the case of prodfsat. The results of proving satisfiability of random formulae with constants (not shown) share the pattern of proving validity with constants.

Overall, we conclude the performance of prodfsat over large input is inferior to that of mNiBLoS. The majority of generated formulae were satisfiable non-tautologies, so in

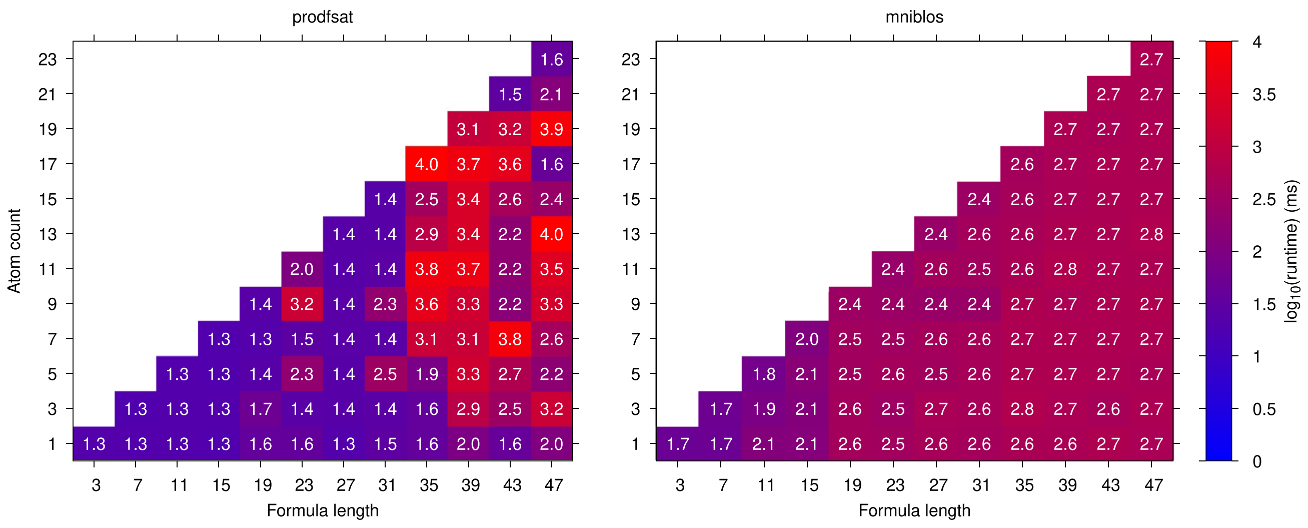

Fig. 5 Runtime of prodfsat (left)and mNiBLoS (right) in decimal-logarithmic scale required to prove the validity of random formulae with constants with increasing length (3–47) and number of atoms

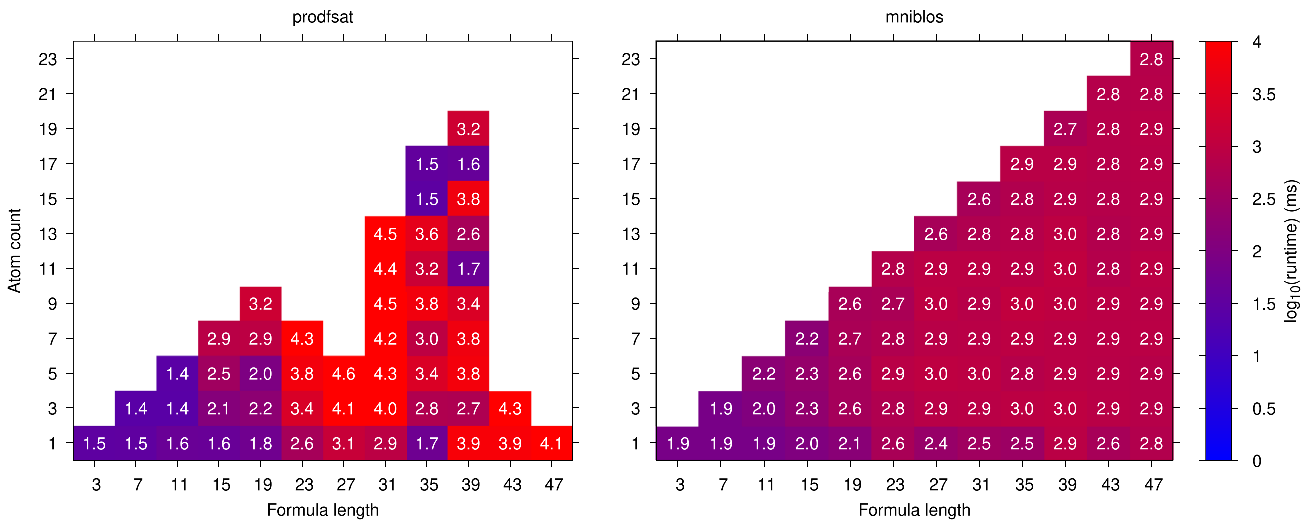

Fig. 6 Runtime of prodfsat (left) and mNiBLoS (right) in decimal-logarithmic scale required to prove the satisfiability of random formulae with increasing length (3–47) and number of atoms. Values exceeding the timeout threshold of 1 min are not shown

the case of proving satisfiability, prodfsat has to traverse all branches of the trichotomous DPLL tree. While we have not yet reached a thorough complexity analysis of the algorithm, the worst case of the branching factor of 3 makes the tree 3^n -exponential. Some cases could be potentially improved by introducing more reduction rules, but the irregularity of randomly generated formulae would probably cause improvements in these tests to be only sporadic.

Hard Instance

Our last experiment also follows Vidal [30, Sect. 4.2] in the evaluation of formulae with fixed structure and quadratically increasing number of atoms. In this section we show that the performance of VAL solving of the following formula with prodfsat is poor in comparison. Then we present an ad hoc reduction rule that improves its performance. Our purpose is not to introduce another DPLL reduction rule that will be used in all future iterations of our work, but rather to show how easily our solver may be extended to help the performance in specific scenarios.

The parameterized version of the problem in question is shown in Eq. (36).

Table 7 Runtime in milliseconds over formulae in Eq. (36) for proving validity

n	prodfsat	prodfsat-opt	mNiBLoS	NiBLoS
2	60.48	29.78	63.93	110.69
3	23.67×10^3	351.50	193.81	857.47
4	66.96×10^6	9009.53	713.60	4192.50
5	Not tested	317.26×10^3	763.19	37.19×10^3
6	Not tested	Not tested	1057.78	Not tested
7	Not tested	Not tested	1226.34	Not tested
8	Not tested	Not tested	1503.56	Not tested
9	Not tested	Not tested	2175.58	Not tested

Table 8 Runtime in milliseconds over formulae in Eq. (36) for proving satisfiability

n	prodfsat	prodfsat-opt	mNiBLoS	NiBLoS
2	20.11	24.07	57.04	57.37
3	19.73	23.00	213.81	189.96
4	20.45	25.31	613.57	1286.33
5	21.81	24.42	753.45	16025.11
6	27.12	25.05	905.96	16989.17
7	24.40	24.50	2449.88	Timed out
8	30.17	26.93	1674.97	Timed out
9	40.43	26.45	12767.42	Timed out
…	…	…	…	…
100	16905.60	17050.62	Timed out	Timed out

The timeout threshold was set to 30 s

$$\bigwedge_{i=1}^{n}(\&_{j=1}^{n}\varphi_{ij}) \to \bigvee_{i=1}^{n}(\&_{j=1}^{n}\varphi_{ij}) \tag{36}$$

We have reconstructed the experiment with the value of the parameter n ranging between 1 and 9 and averaged the performance of all three systems over two measurements for every n. The results for proving validity are shown in Table 7 and satisfiability in Table 8.

The results in the case of validity indicate the inferior performance of prodfsat in comparison with both mNiBLoS and NiBLoS, increasing exponentially to a great degree (the process took 11.6 h to finish with $n = 4$), while the increase of runtime for mNiBLoS is much less steep. There are two reasons for the low performance of prodfsat: (1) the formulae in this experiment are tautologies, so the algorithm has to traverse every branch of the product DPLL tree with the branching factor of 3, which is exacerbated by the quadratic increase of atoms; (2) the clausal forms of the formulae are hard to reduce with the current set of reduction rules.

To demonstrate the feasibility of introducing optimizations to our approach, we have devised a simple additional reduction rule.

$$(\textit{hard problem optimizing rule})\quad \frac{S}{S - \{a \eqcirc b_0^{\beta_0}\ \&\ \dots\ \&\ b_n^{\beta_n}\}};$$
$$a \in \mathit{atoms}(S),\ \text{none of } b_i\ \ 0 \le i \le n,\ \text{occur elsewhere in } S. \tag{37}$$

The reduction rule (37) removes all unit clauses of equality literals between a single atom and a strong conjunction of atoms if all of the atoms of the strong conjunction only occur in this clause. When applied to formulae in Eq. (36), this removes from the theory all clauses that define auxiliary atoms representing strong conjunctions of atoms. After performing this reduction, the clausal theory is equisatisfiable to the original formula, as we do not remove the occurrence of the auxiliary atoms representing the strong conjunctions from the rest of the theory, i.e., we only omit the leaf clauses. In practice, the output of the program with the optimizing rule for $n = 4$ is as follows.

```
 ...
(((*4) = (x30 & x31 & x32 & x33)))
(((*7) = (x20 & x21 & x22 & x23)))
(((*9) = (x00 & x01 & x02 & x03)))
(((*10) = (x10 & x11 & x12 & x13)))
 ...
removing clause *4 = (x30 & x31 & x32 & x33)
removing clause *7 = (x20 & x21 & x22 & x23)
removing clause *9 = (x00 & x01 & x02 & x03)
removing clause *10 = (x10 & x11 & x12 & x13)
```

The runtime of prodfsat with this additional reduction rule is shown in the table as **prodfsat-opt**. As can be seen, mNiBLoS is still vastly superior, but proving the validity of the formula becomes reasonably fast for $n \le 4$.

Proving the satisfiability of formulae in Eq. (36) is much easier for prodfsat than proving validity. The measurements are shown in 8. In this test, prodfsat consistently outperforms mNiBLoS and NiBLoS even without the additional reduction rule. This is because to prove satisfiability, prodfsat does not have to traverse all of the branches of the product DPLL tree—the algorithm stops at the first open branch. The higher complexity of [m]NiBLoS is probably caused by the fact that the SMT solver attempts to find the model—the interpretation of every atom. As the solver is not informed about the nature of the formula, the search space is most likely not well constrained.

Examination of Individual Improvements

In this part we examine how the improvements of the algorithms or implementation of our solver contribute to its runtime performance. We have carried out the experiments from section "Test Set (Batch)" (SAT and VAL over test

Table 9 Runtime in milliseconds with selectively enabled combinations of improvements

Exp.	Opt.	other	pow	dag	pow+dag
batch	other	4494.1	1600.1	2454.4	**1364.7**
	a12	4290.9	1453.4	2281.3	1405.3
	a3	4087.4	1412.8	2163.2	1381.0
	a123	4228.4	1440.5	2238.9	1394.6
hp–val–3	other	2.5×10^5	1.2×10^5	46152.6	22780.9
	a12	2.3×10^5	1.2×10^5	40307.9	22216.5
	a3	2.5×10^5	1.1×10^5	44806.8	22258.5
	a123	2.2×10^5	1.1×10^5	39541.4	**22114.5**
hp–sat–3	other	3.0	2.0	2.1	1.5
	a12	3.1	2.0	2.3	1.4
	a3	3.5	2.0	2.2	1.3
	a123	3.0	2.2	2.0	**1.2**
hp–sat–10	other	215.2	25.2	84.3	13.4
	a12	306.4	22.7	94.0	13.9
	a3	229.5	22.0	79.8	13.3
	a123	293.0	24.8	91.1	**13.0**
hp–sat–100	other	Timed out	27795.2	Timed out	14293.2
	a12	Timed out	27638.5	Timed out	**13365.4**
	a3	Timed out	35102.7	Timed out	15000.9
	a123	Timed out	37908.1	Timed out	15495.7

other represents miscellaneous (most notably cache locality-related) implementation optimizations with all other improvements disabled. *pow* is the avoidance of translating strong conjunctions into order clausal form that are composed only of powers of atoms or constants as per section "Implementation". *dag* is the re-use of subformulae during translation as described in section "Translation into Order Clausal Form" (treating the formula as a directed acyclic graph instead of a tree). *a{123}* is the application of guard propagation rule I, II, and/or III, respectively. *exp* is the experiment the runtime over which was measured in the respective part of the table. hp–(sat/val)–*n* is the experiment performing SAT or VAL on the hard instance in section "Hard Instance" with specified parameter *n*. The timeout for experiment hp–sat–100 was set to 60 s. Minimum values within experiments are shown in bold

formulae run in batch) and section "Hard Instance" (individual SAT and VAL over hard instance) with selectively enabled improvements in a cross-product manner. The measurements of runtime are displayed in Table 9.

The breakdown of runtime shown in Table 9 indicates that two improvements have the highest impact: the avoidance of translating strong conjunctions into order clausal form that are composed only of powers of atoms or constants (*pow* in the table, mentioned in section "Implementation"), and the caching of subformulae during translation (*dag* in the table, described in section "Translation into Order Clausal Form"). The combination of these improvements overall yields the fastest performance. The addition of Guller's admissible DPLL rules (14)–(16) as shown in section "Product DPLL Procedure" improves the runtime only marginally. In some cases, especially in batch formula tests and when solving satisfiability of long formulae (hard instance with $n = 100$), the employment of guard propagation rule III in combination with *pow* and *dag* makes the runtime slightly worse (applying the rule has a cost even if no changes are made), but helps in other experiments. This raises the question whether the addition of admissible rules is suitable for the solver at all. The current version of prodfsat employs these rules because of their ability to produce more compact subtrees, which may improve visualization of the DPLL procedure. However, we will consider their automated selective activation according to the nature of input in future work.

Conclusion and Future Work

In this paper we have presented the improvements to our fuzzy DPLL-based solver for product propositional logic. We have empirically evaluated the current state of our implementation and compared it with our previous work. The results show a considerable increase in performance of SAT and VAL solving, climbing to approximately a four-fold enhancement in our test bench.

More importantly, we have compared the performance of our solver with the existing solvers NiBLoS and mNiBLoS on a set of experiments and obtained the timings of solving SAT and VAL over (1) a fixed set of tests, (2) a formula with parameterized power, (3) randomly generated formulae, and (4) a hard formula instance with a quadratically increasing number of atoms. The results show that our solver excels at (a) formulae short in length, (b) in cases when the DPLL tree can be well-reduced and does not have to be fully traversed (proving SAT of satisfiable and VAL of unsatisfiable formulae), and (c) formulae where the solver leverages its interpretation of product logic. Moreover, even though our solver did not perform well at solving VAL of the hard formula instance, we have demonstrated its advantage of being self-contained by designing and adding a simple reduction rule that downsized the DPLL tree and improved the test results by several orders of magnitude.

To the best of our knowledge, this is the only solution in the group of product propositional fuzzy SAT solvers that generalize classical logic approaches and have a publicly available implementation. However, the current version of our solver does not support intermediate constants and does not yet perform model-finding. We consider these two as the most important features for future addition.

Funding This work was partially supported by the following projects:
– ORBIS, funded by the Slovak Research and Development Agency (SRDA) under contract no. APVV-19-0220.
– KATO, funded by the Slovak VEGA agency under contract no. 1/0778/18.

– TAILOR, funded by the EU Horizon 2020 research and innovation program under Grant Agreement no. 952215.

Availability of data and material Not applicable.

Code availability The source code of prodfsat along with the parts used to conduct experiments is available at the following https://git.uhliarik.com/ivor/prodfsat.

Declarations

Conflict of interest Author Ivor Uhliarik declares that he has no conflict of interest.

Ethical approval This article does not contain any studies with human participants or animals performed by any of the authors.

References

1. Alsinet T, Barroso D, Béjar R, Bou F, Cerami M, Esteva F. On the implementation of a fuzzy DL solver over infinite-valued product logic with SMT solvers. In: Liu W, Subrahmanian VS, Wijsen J (eds) Scalable uncertainty management—7th international conference, SUM 2013, Washington, DC, USA, September 16–18, 2013. Proceedings, Springer, Lecture Notes in Computer Science, vol 8078; 2013. pp. 325–30. https://doi.org/10.1007/978-3-642-40381-1_25.
2. Alviano M, Peñaloza R. Fuzzy answer set computation via satisfiability modulo theories. TPLP. 2015;15(4–5):588–603. https://doi.org/10.1017/S1471068415000241.
3. Ansótegui C, Bofill M, Manyà F, Villaret M. Building automated theorem provers for infinitely-valued logics with satisfiability modulo theory solvers. Fuzzy Sets Syst. 2012;2012:25–30. https://doi.org/10.1109/ISMVL.2012.63.
4. Ansótegui C, Bofill M, Manyá F, Villaret M. Automated theorem provers for multiple-valued logics with satisfiability modulo theory solvers. Fuzzy Sets Syst. 2016;292:32–48.
5. Baaz M, Hájek P, Švejda D, Krajíček J. Embedding logics into product logic. Stud Log. 1998;61(1):35–47. https://doi.org/10.1023/A:1005026229560.
6. Bobillo F, Straccia U. A fuzzy description logic with product t-norm. In: 2007 IEEE international fuzzy systems conference; 2007a. pp. 1–6. https://doi.org/10.1109/FUZZY.2007.4295443.
7. Bobillo F, Straccia U. A fuzzy description logic with product t-norm. In: Fuzzy systems conference, 2007. FUZZ-IEEE 2007. IEEE International, IEEE; 2007b. pp. 1–6.
8. Brys T, Drugan MM, Bosman PA, De Cock M, Nowé A. Solving satisfiability in fuzzy logics by mixing CMA-ES. In: Proceedings of the 15th annual conference on genetic and evolutionary computation, ACM, New York, NY, USA, GECCO '13; 2013. pp. 1125–1132. https://doi.org/10.1145/2463372.2463510.
9. Béjar R, Alsinet T, Bou F, Barroso D, Cerami M, Esteva F. On the implementation of a fuzzy DL solver over infinite-valued product logic with SMT solvers. Berlin: Springer; 2013. p. 8078. https://doi.org/10.1007/978-3-642-40381-1_25.
10. Davis M, Logemann G, Loveland D. A machine program for theorem-proving. Commun ACM. 1962;5(7):394–7. https://doi.org/10.1145/368273.368557.
11. de Moura L, Bjørner N. Z3: An efficient SMT solver. In: Ramakrishnan CR, Rehof J, editors. Tools and algorithms for the construction and analysis of systems. Heidelberg: Springer; 2008. p. 337–40.
12. Guller D. A DPLL procedure for the propositional product logic. In: Proceedings of the 5th international joint conference on computational intelligence—Volume 1: FCTA, (IJCCI 2013), INSTICC, SciTePress; 2013. pp. 213–224. https://doi.org/10.5220/0004557402130224.
13. Guller D. An order hyperresolution calculus for Gödel logic with truth constants and equality, strict order, delta. In: 2015 7th international joint conference on computational intelligence (IJCCI); 2015. pp. 31–46.
14. Guller D. Hyperresolution for propositional product logic. In: Guervós JJM, Melício F, Cadenas JM, Dourado A, Madani K, Ruano AEB, Filipe J (eds) Proceedings of the 8th international joint conference on computational intelligence, IJCCI 2016, Volume 2: FCTA, Porto, Portugal, November 9-11, 2016, SciTePress; 2016. pp. 30–41. https://doi.org/10.5220/0006044300300041.
15. Guller D. Technical foundations of a DPLL-based SAT solver for propositional gödel logic. IEEE Trans Fuzzy Syst. 2018;26(1):84–100. https://doi.org/10.1109/TFUZZ.2016.2637374.
16. Guller D. Hyperresolution for Gödel logic with truth constants. Fuzzy Sets Syst. 2019;363:1–65. https://doi.org/10.1016/j.fss.2018.09.008.
17. Guller D. Hyperresolution for propositional product logic with truth constants. Cham: Springer International Publishing; 2019. p. 197–220. https://doi.org/10.1007/978-3-319-99283-9_10.
18. Hähnle R. Many-valued logic and mixed integer programming. Ann Math Artif Intell. 1994;12(3):231–63. https://doi.org/10.1007/BF01530787.
19. Hájek P. Metamathematics of fuzzy logic. In: Trends in logic. Berlin: Springer; 2001.
20. Haller M, Haberl R, Carbonell D, Philippen D, Frank E. Sol-heap-solar and heat pump combisystems. Institut für Solartechnik SPF, Hochschule für Technik HSR, Rapperswil, Switzerland, Report Contract No SI/500494-02. 2014.
21. Janssen J, Schockaert S, Vermeir D, De Cock M. Answer set programming for continuous domains: a fuzzy logic approach. In: Atlantis computational intelligence systems. Amsterdam: Atlantis Press; 2012.
22. Lukasiewicz T, Straccia U. Managing uncertainty and vagueness in description logics for the semantic web. J Web Semant. 2008;6(4):291–308. https://doi.org/10.1016/j.websem.2008.04.001 (**Semantic web challenge 2006/2007**).
23. Mostert PS, Shields AL. On the structure of semigroups on a compact manifold with boundary. Ann Math. 1957;65(1):117–43 (http://www.jstor.org/stable/1969668).
24. Plaisted DA, Greenbaum S. A structure-preserving clause form translation. J Symb Comput. 1986;2(3):293–304.
25. Réseau de Transport d'Électricité. RTE daily energy load data. 2022. https://www.services-rte.com/en/view-data-published-by-rte/daily-load-curves.html.
26. Tseitin GS. On the complexity of derivation in propositional calculus. Berlin, Heidelberg: Springer; 1983. p. 466–83. https://doi.org/10.1007/978-3-642-81955-1_28.
27. Uhliarik I. Foundations of a DPLL-based solver for fuzzy answer set programs. Cham: Springer International Publishing; 2019. p. 99–117. https://doi.org/10.1007/978-3-030-16469-0_6.
28. Uhliarik I (2020) The implementation of a product fuzzy DPLL solver. In: Guervós JJM, Garibaldi JM, Wagner C, Bäck T, Madani K, Warwick K (eds) Proceedings of the 12th international joint conference on computational intelligence, IJCCI 2020, Budapest, Hungary, November 2-4, 2020, SCITEPRESS; 2020. pp. 252–63. https://doi.org/10.5220/0010148802520263.
29. Van Nieuwenborgh D, De Cock M, Vermeir D. An introduction to fuzzy answer set programming. Ann Math Artif Intell. 2007;50(3):363–88. https://doi.org/10.1007/s10472-007-9080-3.

30. Vidal A. MNiBLoS: A SMT-based solver for continuous t-norm based logics and some of their modal expansions. Inf Sci. 2016;372:709–30. https://doi.org/10.1016/j.ins.2016.08.072.
31. Vidal A. NiBLoS: a nice BL-logics solver. Master's Thesis, Universitat de Barcelona. 2012.