Integrating SAT Solvers with Math Reasoners: Foundations and Basic Algorithms

Roberto Sebastiani[1,2]

[1] Dept. of Mathematics, University of Trento, 38050 Povo, (TN), Italy.
[2] ITC-IRST, 38050 Povo (TN), Italy.
rseba@science.unitn.it

Abstract. In the last years we have witnessed an impressive advance in the efficiency of SAT techniques, which has brought large previously intractable problems at the reach of state-of-the-art solvers. Unfortunately, simple boolean expressions are not expressive enough for representing many real-world problems, which require handling also integer or real values and operators. On the other hand, mathematical reasoners, like computer-algebra systems or constraint solvers, cannot handle efficiently problems involving heavy boolean search, or do not handle them at all. In this paper we present the foundations and the basic algorithms for a new class of procedures for solving boolean combinations of mathematical propositions, which combine SAT solvers and mathematical reasoners.

1 Motivation and goals

In the last years we have witnessed an impressive advance in the efficiency of SAT techniques, which has brought large previously intractable problems at the reach of state-of-the-art solvers. As a consequence, some hard real-world problems have been successfully solved by encoding them into SAT. Propositional planning [15] and boolean model-checking [4] are among the best achievements.

Unfortunately, simple boolean expressions are not expressive enough for representing many real-world problems. For example, problem domains like temporal reasoning, resource planning, verification of systems with numerical data values or of timed systems, often require handling also constraints on integer or real-valued quantities (see, e.g., [1, 18, 17, 19]). Moreover, some problem domains like model checking often require an explicit representation of integers and arithmetic operators, which simple boolean expressions are not good at representing efficiently (see, e.g., [17, 14]). On the other hand, mathematical reasoners, like computer-algebra systems or constraint solvers, cannot handle efficiently problems involving heavy boolean search, or do not handle them at all.

In 1996 we have proposed a new general approach, called *SAT based*, for building domain-specific decision procedures on top of SAT solvers [10, 12]. The basic idea was to decompose the search into two orthogonal components, one purely propositional component and one "boolean-free" domain-specific component and to use a (modified) Davis Putnam Longemann Loveland (DPLL) SAT solver [6] for the former and a pure domain-specific procedure for the latter. So far the SAT based approach proved very effective in various problem domains like, e.g., modal and description logics [12, 8], temporal reasoning [1], resource planning [18].

In this paper we present the foundations and the basic algorithms for a new class of procedures for solving boolean combinations of mathematical propositions, which combine SAT solvers and mathematical reasoners. The ultimate goal is to develop solvers able to handle complex verification problems like those hinted above. The paper is structured as follows. In Section 2 we describe formally the problem we are addressing. In Section 3 we present the logic framework on which the procedures are based. In Section 4 we present a generalized search procedure, we prove its basic properties and introduce some efficiency issues. In Section 5 we present a basic procedure based on DPLL, we prove its basic properties and suggest some algorithmic improvements. In Section 6 we describe some related work.

2 The problem

We address the problem of checking the satisfiability of boolean combinations of primitive and mathematical propositions. Let $\mathcal{D}$ be the domain of either integer numbers $\mathbb{Z}$ or real numbers $\mathbb{R}$, with the respective set $\mathcal{OP}_\mathcal{D}$ of arithmetical operators $\{+, -, \cdot, /, mod\}$ or $\{+, -, \cdot, /\}$ respectively. Let $\{\bot, \top\}$ denote the $false$ and $true$ boolean values. Given the standard boolean connectives $\{\neg, \wedge\}$ and math operators $\{=, \neq, >, <, \geq, \leq\}$, let $\mathcal{A} = \{A_1, A_2, \ldots\}$ be a set of primitive propositions, let $\mathcal{C} = \{c_1, c_2, \ldots\}$ and $\mathcal{V} = \{v_1, v_2, \ldots\}$ respectively be a set of numerical constants and variables over the domain $\mathcal{D}$.

We call $\mathcal{D}$-*terms* the mathematical expressions built on constants, variables and arithmetical operators over $\mathcal{D}$:

- a constant $c_i \in \mathcal{C}$ is a $\mathcal{D}$-term;
- a variable $v_i \in \mathcal{V}$ is a $\mathcal{D}$-term;
- if t_1 is a $\mathcal{D}$-term, then $-t_1$ is a $\mathcal{D}$-term;
- if t_1, t_2 are $\mathcal{D}$-terms, then $(t_1 \otimes t_2)$ is a $\mathcal{D}$-term, $\otimes \in \mathcal{OP}_\mathcal{D}$.

For instance, $((v_1 + 5.0)/(2.0 \cdot v_3))$ is a $\mathbb{R}$-term if v_1 and v_2 are real variables, whilst $((2 \cdot v_1)\ mod\ (3 - v_2))$ is an $\mathbb{Z}$-term if v_1 and v_2 are integer.

We call $\mathcal{D}$-*formulas* the mathematical formulas built on primitive propositions, $\mathcal{D}$-terms, operators and boolean connectives:

- a primitive proposition $A_i \in \mathcal{A}$ is a $\mathcal{D}$-formula;
- if t_1, t_2 are $\mathcal{D}$-terms, then $(t_1 \bowtie t_2)$ is a $\mathcal{D}$-formula, $\bowtie \in \{=, \neq, >, <, \geq, \leq\}$;
- if φ_1 is a $\mathcal{D}$-formula, then $\neg\varphi_1$ is a $\mathcal{D}$-formula;
- if φ_1, φ_2 are $\mathcal{D}$-formulas, then $(\varphi_1 \wedge \varphi_2)$ is a $\mathcal{D}$-formula.

For instance, $A_1 \wedge ((v_1 + 5.0) \leq (2.0 \cdot v_3))$ is a $\mathbb{R}$-formula if v_1 and v_2 are real variables, while $A_2 \wedge \neg(((2 \cdot v_1)\ mod\ v_2) > 5)$ is an $\mathbb{Z}$-formula if v_1 and v_2 are integer. [1]

Notationally, we use the downcase letters $t, t_1, \ldots$ to denote $\mathcal{D}$-terms, and the Greek letters $\alpha, \beta, \varphi, \psi$ to denote $\mathcal{D}$-formulas. We use the standard abbreviations, that is: "$\varphi_1 \vee \varphi_2$" for "$\neg(\neg\varphi_1 \wedge \neg\varphi_2)$", "$\varphi_1 \rightarrow \varphi_2$" for "$\neg(\varphi_1 \wedge \neg\varphi_2)$", "$\varphi_1 \leftrightarrow \varphi_2$" for "$\neg(\varphi_1 \wedge \neg\varphi_2) \wedge \neg(\varphi_2 \wedge \neg\varphi_1)$", "$T$" for any valid formula, and "F" for "$\neg T$". We call $Terms_\mathcal{D}$ and $Formulas_\mathcal{D}$ the set of

[1] The assumption that the domain is the whole $\mathbb{Z}$ or $\mathbb{R}$ is not restrictive, as we can restrict the domain of any variable v_i at will by adding to the formula constraints like, e.g., $(v_1 \neq 0.0)$, $(v_1 \leq 5.0)$, etc.

$\mathcal{D}$-terms and $\mathcal{D}$-formulas respectively. When this does not cause ambiguities, we use the associativity and precedence rules of arithmetical operators to simplify the appearance of $\mathcal{D}$-terms; e.g, we write "$(c_1(v_2 - v_1) - c_1 v_3 + c_3 v_4)$" instead of "$(((c_1 \cdot (v_2 - v_1)) - (c_1 \cdot v_3)) + (c_3 \cdot v_4))$".

We call $\mathcal{D}$-*interpretation* a map $\mathcal{I}$ which assigns $\mathcal{D}$ values and boolean values to $\mathcal{D}$-terms and $\mathcal{D}$-formulas respectively and preserves constants and arithmetical operators, [2] that is, $\mathcal{I} =_{def} \langle \mathcal{I}_\mathcal{D}, \mathcal{I}_b \rangle$, s.t. $\mathcal{I}_b : \mathcal{A} \longmapsto \{\top, \bot\}$ and $\mathcal{I}_\mathcal{D} : Terms_\mathcal{D} \longmapsto \mathcal{D}$, and

- $\mathcal{I}_b(A_i) \in \{\top, \bot\}$, for every $A_i \in \mathcal{A}$;
- $\mathcal{I}_\mathcal{D}(c_i) = c_i$, for every $c_i \in \mathcal{C}$;
- $\mathcal{I}_\mathcal{D}(v_i) \in \mathcal{D}$, for every $v_i \in \mathcal{V}$;
- $\mathcal{I}_\mathcal{D}(t_1 \otimes t_2) = (\mathcal{I}_\mathcal{D}(t_1) \otimes \mathcal{I}_\mathcal{D}(t_2))$, for all $\mathcal{D}$-terms t_1, t_2 and $\otimes \in \mathcal{OP}_\mathcal{D}$.

The binary relation $\models_\mathcal{D}$ between a $\mathcal{D}$-interpretation $\mathcal{I}$ and a $\mathcal{D}$-formula φ, written "$\mathcal{I} \models_\mathcal{D} \varphi$" ("$\mathcal{I}$ *satisfies* φ in $\mathcal{D}$" or "$\mathcal{I}$ $\mathcal{D}$-satisfies φ") is defined as follows:

$$
\begin{array}{ll}
\mathcal{I} \models_\mathcal{D} A_i, \ A_i \in \mathcal{A} & \iff \mathcal{I}_b(A_i) = \top; \\
\mathcal{I} \models_\mathcal{D} (t_1 \bowtie t_2), \bowtie \in \{=, \neq, >, <, \geq, \leq\} & \iff \mathcal{I}_\mathcal{D}(t_1) \bowtie \mathcal{I}_\mathcal{D}(t_2); \\
\mathcal{I} \models_\mathcal{D} \neg\varphi_1 & \iff \mathcal{I} \not\models_\mathcal{D} \varphi_1; \\
\mathcal{I} \models_\mathcal{D} (\varphi_1 \wedge \varphi_2) & \iff \mathcal{I} \models_\mathcal{D} \varphi_1 \textit{ and } \mathcal{I} \models_\mathcal{D} \varphi_2.
\end{array}
$$

We say that a $\mathcal{D}$-formula φ is *satisfiable in* $\mathcal{D}$ ($\mathcal{D}$-satisfiable) if and only if there exist a $\mathcal{D}$-interpretation $\mathcal{I}$ s.t. $\mathcal{I} \models_\mathcal{D} \varphi$. E.g., if $\mathcal{D} = \mathbb{R}$, then $A_1 \rightarrow ((v_1 + 2v_2) \leq 4.5)$ is $\mathbb{R}$-satisfied by an $\mathbb{R}$-interpretation $\mathcal{I}$ s.t. $\mathcal{I}_b(A_1) = \top$, $\mathcal{I}_\mathcal{D}(v_1) = 1.1$, and $\mathcal{I}_\mathcal{D}(v_2) = 0.6$. For every φ_1 and φ_2, we say that $\varphi_1 \models_\mathcal{D} \varphi_2$ if and only if $\mathcal{I} \models_\mathcal{D} \varphi_2$ for every $\mathcal{I}$ s.t. $\mathcal{I} \models_\mathcal{D} \varphi_1$. We also say that $\models_\mathcal{D} \varphi$ (φ is $\mathcal{D}$-*valid*) if and only if $\mathcal{I} \models_\mathcal{D} \varphi$ for every $\mathcal{I}$. It is easy to verify that $\varphi_1 \models_\mathcal{D} \varphi_2$ if and only if $\models_\mathcal{D} \varphi_1 \rightarrow \varphi_2$, and that $\models_\mathcal{D} \varphi$ iff $\neg\varphi$ is $\mathcal{D}$-unsatisfiable.

As standard boolean formulas are a strict subcase of $\mathcal{D}$-formulas, it follows trivially that $\mathcal{D}$-satisfiability is "at least as hard" as boolean satisfiability.

Proposition 1. *The problem of deciding the $\mathcal{D}$-satisfiability of a $\mathcal{D}$-formula φ is NP-hard.*

Thus, deciding $\mathcal{D}$-satisfiability is computationally very expensive. The complexity upper bound may depend on the kind of mathematical problems we are dealing. For instance, if we are dealing with (boolean combinations of) arithmetical expressions over bounded integers, then for every $\mathcal{I}$ we can verify $\mathcal{I} \models_\mathcal{D} \varphi$ in a polynomial amount of time, and thus the problem is NP-complete.

3 The formal framework

Definition 1. *We call* **atom** *any $\mathcal{D}$-formula that cannot be decomposed propositionally, that is, any $\mathcal{D}$-formula whose main connective is not a boolean operator. A* **literal** *is either an atom (a* **positive** *literal) or its negation (a* **negative** *literal).*

[2] Here we make a little abuse of notation with the constants and the operators in $\mathcal{OP}_\mathcal{D}$. In fact, e.g., we denote by the same symbol "+" both the language symbol in $\mathcal{I}_\mathcal{D}(t_1 + t_2)$ and for the arithmetic operator in $(\mathcal{I}_\mathcal{D}(t_1) + \mathcal{I}_\mathcal{D}(t_2))$. Same discourse holds for the constants $c_i \in \mathcal{C}$ and also for the operators $\{=, \neq, >, <, \geq, \leq\}$.

Examples of literals are, A_1, $\neg A_2$, $(v_1 + 5.0 \leq 2.0v_3)$, $\neg((2v_1 \ mod \ v_2) > 5)$. If l is a negative literal $\neg\psi$, then by "$\neg l$" we conventionally mean ψ rather than $\neg\neg\psi$. We denote by $Atoms(\varphi)$ the set of atoms in φ. Notice that $Atoms(\neg\varphi) = Atoms(\varphi)$ and that $Atoms(\varphi_1 \wedge \varphi_2) = Atoms(\varphi_1) \cup Atoms(\varphi_2)$.

Definition 2. *We call a* **total truth assignment** μ *for a $\mathcal{D}$-formula φ a set*

$$\mu = \{\alpha_1, \ldots, \alpha_N, \neg\beta_1, \ldots, \neg\beta_M, A_1, \ldots, A_R, \neg A_{R+1}, \ldots, \neg A_S\}, \tag{1}$$

such that every atom in $Atoms(\varphi)$ occurs as a either positive or negative literal in μ. A **partial truth assignment** *μ for φ is a subset of a total truth assignment for φ. If $\mu_2 \subseteq \mu_1$, then we say that μ_1* **extends** *μ_2 and that μ_2* **subsumes** *μ_1.*

A total truth assignment μ is interpreted as a truth value assignment to all the atoms of φ: $\alpha_i \in \mu$ means that α_i is assigned to $\top$, $\neg\beta_i \in \mu$ means that β_i is assigned to $\bot$. Syntactically identical instances of the same atom are always assigned identical truth values; syntactically different atoms, e.g., $(t_1 \geq t_2)$ and $(t_2 \leq t_1)$, are treated differently and may thus be assigned different truth values.

Notationally, we use the Greek letters μ, η to represent truth assignments. Furthermore, we often write a truth assignment like (1) as the conjunction of its elements:

$$\mu = \bigwedge_i \alpha_i \wedge \bigwedge_j \neg\beta_j \wedge \gamma, \tag{2}$$

where $\gamma = \bigwedge_{k=1}^{R} A'_k \ \wedge \bigwedge_{h=R+1}^{S} \neg A'_h$ is a conjunction of propositional literals. Thus, we say that an assignment is $\mathcal{D}$-satisfiable meaning that its corresponding $\mathcal{D}$-formula (2) is $\mathcal{D}$-satisfiable. From now on we indifferently represent a truth assignment as a set (1) or as a formula (2).

Definition 3. *We say that a total truth assignment μ for φ* **propositionally $\mathcal{D}$-satisfies** *φ, written $\mu \models_p \varphi$, if and only if it makes φ evaluate to $\top$, that is, for all sub-formulas φ_1, φ_2 of φ:*

$$\begin{array}{ll} \mu \models_p \varphi_1, \ \varphi_1 \in Atoms(\varphi) & \Longleftrightarrow \varphi_1 \in \mu; \\ \mu \models_p \neg\varphi_1 & \Longleftrightarrow \mu \not\models_p \varphi_1; \\ \mu \models_p \varphi_1 \wedge \varphi_2 & \Longleftrightarrow \mu \models_p \varphi_1 \text{ and } \mu \models_p \varphi_2. \end{array}$$

We say that a partial truth assignment μ **propositionally $\mathcal{D}$-satisfies** *φ if and only if all the total truth assignments for φ which extend μ propositionally $\mathcal{D}$-satisfy φ.*

From now on, if not specified, when dealing with propositional $\mathcal{D}$-satisfiability we do not distinguish between total and partial assignments.

We say that φ is *propositionally $\mathcal{D}$-satisfiable* if and only if there exist an assignment μ s.t. $\mu \models_p \varphi$. Intuitively, if we consider a $\mathcal{D}$-formula φ as a propositional formulas in its atoms, then $\models_p$ is the standard satisfiability in propositional logic. Thus, for every φ_1 and φ_2, we say that $\varphi_1 \models_p \varphi_2$ if and only if $\mu \models_p \varphi_2$ for every μ s.t. $\mu \models_p \varphi_1$. We also say that $\models_p \varphi$ (φ is *propositionally valid*) if and only if $\mu \models_p \varphi$ for every assignment μ for φ. It is easy to verify that $\varphi_1 \models_p \varphi_2$ if and only if $\models_p \varphi_1 \rightarrow \varphi_2$, and that $\models_p \varphi$ iff $\neg\varphi$ is propositionally $\mathcal{D}$-unsatisfiable.

Notice that $\models_p$ is stronger than $\models_{\mathcal{D}}$, that is, if $\varphi_1 \models_p \varphi_2$, then $\varphi_1 \models_{\mathcal{D}} \varphi_2$, but not vice versa. E.g., $(v_1 \leq v_2) \wedge (v_2 \leq v_3) \models_{\mathcal{D}} (v_1 \leq v_3)$, but $(v_1 \leq v_2) \wedge (v_2 \leq v_3) \not\models_p (v_1 \leq v_3)$.

Example 1. Consider the following $\mathcal{D}$-formula φ:

$$\begin{array}{l}\varphi = \{\underline{\neg(2v_2 - v_3 > 2)} \vee A_1\} \wedge \\ \quad \{\underline{\neg A_2} \vee (2v_1 - 4v_5 > 3)\} \wedge \\ \quad \{\underline{(3v_1 - 2v_2 \leq 3)} \vee A_2\} \quad \wedge \\ \quad \{\neg(2v_3 + v_4 \geq 5) \vee \underline{\neg(3v_1 - v_3 \leq 6)} \vee \neg A_1\} \quad \wedge \\ \quad \{A_1 \vee \underline{(3v_1 - 2v_2 \leq 3)}\} \quad \wedge \\ \quad \{\underline{(v_2 - v_4 \leq 6)} \vee (v_5 = 5 - 3v_4) \vee \neg A_1\} \quad \wedge \\ \quad \{A_1 \vee \underline{(v_3 = 3v_5 + 4)} \vee A_2\}\end{array}$$

The partial truth assignment given by the underlined literals above is:

$$\begin{array}{lll}\mu = (v_2 - v_4 \leq 6) \quad \wedge (v_3 = 3v_5 + 4) \quad \wedge (3v_1 - 2v_2 \leq 3) \wedge & [\bigwedge_i \alpha_i] \\ \quad \neg(2v_2 - v_3 > 2) \wedge \neg(3v_1 - v_3 \leq 6) \wedge & [\bigwedge_j \neg\beta_j] \\ \quad \neg A_2. & [\gamma]\end{array}$$

μ is a partial assignment which propositionally $\mathcal{D}$-satisfies φ, as it sets to true one literal of every disjunction in φ, so that every total assignment which extends μ propositionally $\mathcal{D}$-satisfies φ. Notice that μ is not $\mathcal{D}$-satisfiable, as its sub-assignment

$$(3v_1 - 2v_2 \leq 3) \wedge \neg(2v_2 - v_3 > 2) \wedge \neg(3v_1 - v_3 \leq 6) \tag{3}$$

(rows 3, 1 and 4 in φ) does not have any $\mathcal{D}$-satisfying interpretation. $\diamond$

Definition 4. *We say that a collection $\mathcal{M} = \{\mu_1, \ldots, \mu_n\}$ of partial assignments propositionally $\mathcal{D}$-satisfying φ is* **complete** *if and only if*

$$\models_p \varphi \leftrightarrow \bigvee_j \mu_j. \tag{4}$$

$\mathcal{M}$ is complete in the sense that, for every total assignment η s.t. $\eta \models_p \varphi$, there exists $\mu_j \in \mathcal{M}$ s.t. $\mu_j \subseteq \eta$. Therefore $\mathcal{M}$ is a compact representation of the whole set of total assignments propositionally $\mathcal{D}$-satisfying φ. Notice that $\models_{\mathcal{D}} \varphi \leftrightarrow \bigvee_j \mu_j$, as $\models_p$ is stronger than $\models_{\mathcal{D}}$.

Given an interpretation $\mathcal{I}$ and a $\mathcal{D}$-formula φ, we define the total truth assignment for φ **induced** by $\mathcal{I}$, written $[\mathcal{I}]^\varphi$, as:

$$[\mathcal{I}]^\varphi =_{def} \{\varphi_i \mid \varphi_i \in \mathit{Atoms}(\varphi) \text{ and } \mathcal{I} \models_{\mathcal{D}} \varphi_i\} \bigcup \{\neg\varphi_i \mid \varphi_i \in \mathit{Atoms}(\varphi) \text{ and } \mathcal{I} \not\models_{\mathcal{D}} \varphi_i\}.$$

Intuitively, $[\mathcal{I}]^\varphi$ is the total truth assignment given by the truth values implicitly assigned by $\mathcal{I}$ to the atoms of φ.

Example 2. Consider the $\mathcal{D}$-formula φ in Example 1 and the interpretation $\mathcal{I}$ s.t. $\mathcal{I}_b(A_1) = \top$, $\mathcal{I}_b(A_2) = \bot$, $\mathcal{I}_{\mathcal{D}}(v_1) = \mathcal{I}_{\mathcal{D}}(v_2) = \mathcal{I}_{\mathcal{D}}(v_3) = \mathcal{I}_{\mathcal{D}}(v_4) = \mathcal{I}_{\mathcal{D}}(v_5) = 0$. Then the induced assignment $[\mathcal{I}]^\varphi$ is

$$\begin{array}{l}\{\ (3v_1 - 2v_2 \leq 3), (3v_1 - v_3 \leq 6), (v_2 - v_4 \leq 6), \neg(2v_2 - v_3 > 2), \\ \quad \neg(2v_1 - 4v_5 > 3), \neg(2v_3 + v_4 \geq 5), \neg(v_5 = 5 - 3v_4), \neg(v_3 = 3v_5 + 4), \\ \quad A_1, \neg A_2\ \} \qquad \diamond.\end{array}$$

The following properties follow straightforwardly from the definition of $[\mathcal{I}]^{\varphi}$.

Property 1. $\mathcal{I} \models_{\mathcal{D}} [\mathcal{I}]^{\varphi}$, for every φ.

Property 2. $[\mathcal{I}]^{\mu} = \mu$, for every truth assignment μ.

Property 3. $[\mathcal{I}]^{\varphi} = [\mathcal{I}]^{\neg\varphi}$, for every φ.

Property 4. $[\mathcal{I}]^{\varphi} \models_p \varphi \iff [\mathcal{I}]^{\varphi\wedge\psi} \models_p \varphi$, for every φ and ψ.

Lemma 1. $\mathcal{I} \models_{\mathcal{D}} \varphi \iff [\mathcal{I}]^{\varphi} \models_p \varphi$.

Proof. By induction on φ.

- atom φ: $[\mathcal{I}]^{\varphi}$ is $\{\varphi\}$. Then the thesis holds for the definition of $\models_p$.
- $\varphi = \neg\varphi_1$:

$$\begin{array}{ll} \mathcal{I} \models_{\mathcal{D}} \neg\varphi_1 & \iff \text{[Definition of } \models_{\mathcal{D}}\text{]} \\ \mathcal{I} \not\models_{\mathcal{D}} \varphi_1 & \iff \text{[Ind. Hypothesis]} \\ [\mathcal{I}]^{\varphi_1} \not\models_p \varphi_1 & \iff \text{[Property 3]} \\ [\mathcal{I}]^{\neg\varphi_1} \not\models_p \varphi_1 & \iff \text{[Definition of } \models_p\text{]} \\ [\mathcal{I}]^{\neg\varphi_1} \models_p \neg\varphi_1. & \end{array}$$

- $\varphi = \varphi_1 \wedge \varphi_2$:

$$\begin{array}{llll} \mathcal{I} \models_{\mathcal{D}} \varphi_1 \wedge \varphi_2 & & & \iff \text{[Definition of } \models_{\mathcal{D}}\text{]} \\ \mathcal{I} \models_{\mathcal{D}} \varphi_1 & \textit{and} & \mathcal{I} \models_{\mathcal{D}} \varphi_2 & \iff \text{[Ind. Hypothesis]} \\ [\mathcal{I}]^{\varphi_1} \models_p \varphi_1 & \textit{and} & [\mathcal{I}]^{\varphi_2} \models_p \varphi_2 & \iff \text{[Property 4]} \\ [\mathcal{I}]^{\varphi_1\wedge\varphi_2} \models_p \varphi_1 & \textit{and} & [\mathcal{I}]^{\varphi_1\wedge\varphi_2} \models_p \varphi_2 & \iff \text{[Definition of } \models_p\text{]} \\ [\mathcal{I}]^{\varphi_1\wedge\varphi_2} \models_p \varphi_1 \wedge \varphi_2. & & & \end{array}$$

□

As a consequence of Lemma 1 we have the following result.

Theorem 1. *A $\mathcal{D}$-formula φ is $\mathcal{D}$-satisfiable if and only if there exists a $\mathcal{D}$-satisfiable total truth assignment μ s.t. $\mu \models_p \varphi$.*

Proof.

- If: $\mu \models_p \varphi$ and $\mathcal{I} \models_{\mathcal{D}} \mu$, for some $\mathcal{I}$. Then, for every $\varphi_1, \varphi_2 \in \mathit{Atoms}(\varphi)$, $\varphi_1 \in \mu \Longrightarrow \mathcal{I} \models_{\mathcal{D}} \varphi_1$ and $\neg\varphi_2 \in \mu \Longrightarrow \mathcal{I} \not\models_{\mathcal{D}} \varphi_2$. Therefore, from the definition of $\models_{\mathcal{D}}$, it trivially follows that $\mathcal{I} \models_{\mathcal{D}} \varphi$.
- Only if: $\mathcal{I} \models_{\mathcal{D}} \varphi$, for some $\mathcal{I}$. Let $\mu = [\mathcal{I}]^{\varphi}$. Then $\mathcal{I} \models_{\mathcal{D}} \mu$ (Property 1) and $\mu \models_p \varphi$ (Lemma 1). □

Example 3. Consider the $\mathcal{D}$-formula φ of Example 1 and the $\mathcal{D}$-interpretation $\mathcal{I}$ with the induced assignment $[\mathcal{I}]^{\varphi}$ of Example 2. It is easy to verify that $[\mathcal{I}]^{\varphi} \models_p \varphi$. Thus φ is $\mathcal{D}$-satisfiable. ◇

```
boolean MATH-SAT(formula φ, assignment & μ, interpretation & I)
    do
        μ := Next(ASSIGN_ENUMERATOR(φ)) /* next in {μ_1, ..., μ_n} */
        if (μ ≠ Null)
            I :=MATHSERVICE_PROVIDER(μ);
    while ((I = Null) and (μ ≠ Null))
    if (I ≠ Null)
    then return True;                    /* a D-satisfiable assignment found */
    else return False;                   /* no D-satisfiable assignment found */
```

Fig. 1. Schema of the generalized search procedure for $\mathcal{D}$-satisfiability.

4 A generalized search procedures

Theorem 1 allows us to split the notion of $\mathcal{D}$-satisfiability of φ into two orthogonal components:

- a *purely boolean* component, consisting in the existence of a propositional model for φ;
- a *purely mathematical* component, consisting in the existence of a $\mathcal{D}$-interpretation for a set of atomic (possibly negated) mathematical propositions.

These two aspects are handled, respectively, by a *truth assignment enumerator* and by a *mathematical service provider*.

Definition 5. *We call a* **truth assignment enumerator** *a total function* ASSIGN_ENUMERATOR *which takes as input a $\mathcal{D}$-formula φ and returns a complete collection $\{\mu_1, \ldots, \mu_n\}$ of assignments $\mathcal{D}$-satisfying φ.*

Definition 6. *We call a* **mathematical service provider** *a total function* MATHSERVICE_PROVIDER *which takes as input a set of (possibly negated) atomic $\mathcal{D}$-formulas μ and returns a $\mathcal{D}$-interpretation satisfying μ, or Null if there is none.*

The general schema of a search procedure for $\mathcal{D}$-satisfiability is reported in Figure 1. MATH-SAT takes as input a formula φ and (by reference) an initially empty assignment μ and an initially null $\mathcal{D}$-interpretation $\mathcal{I}$. For every assignment μ in the collection $\{\mu_1, .., \mu_n\}$ generated by ASSIGN_ENUMERATOR(φ), MATH-SAT invokes MATHSERVICE_PROVIDER over μ, which either returns a $\mathcal{D}$-interpretation satisfying μ, or *Null* if there is none. This is done until either one $\mathcal{D}$-satisfiable assignment is found, or no more assignments are available in $\{\mu_1, ..., \mu_n\}$. In the former case φ is $\mathcal{D}$-satisfiable, in the later case it is not.

It is very important to notice that the search procedure schema of Figure 1 is completely independent on the kind of mathematical domain we are addressing, once we have a mathematical service provider for it.

4.1 Termination, correctness and completeness

MATH-SAT performs at most n loops. Thus, if every call to MATHSERVICE_PROVIDER terminates, then MATH-SAT terminates.

We show that, if MATHSERVICE_PROVIDER is correct and complete, then MATH-SAT is correct and complete.

Lemma 2. *Let φ be a $\mathcal{D}$-formula and let $\mathcal{M} = \{\mu_1, \ldots, \mu_n\}$ be a complete collection of truth assignments $\mathcal{D}$-satisfying φ. Then, for every $\mathcal{D}$-interpretation $\mathcal{I}$, $\mathcal{I} \models_{\mathcal{D}} \varphi$ if and only if $\mathcal{I} \models_{\mathcal{D}} \mu_j$ for at least one $\mu_j \in \mathcal{M}$.*

Proof.

- If: $\mu_j \models_p \varphi$ and $\mathcal{I} \models_{\mathcal{D}} \mu_j$. Let μ'_j be the total assignment extending μ_j s.t. $\mathcal{I} \models_{\mathcal{D}} \mu'_j$. Then $\mu'_j \models_p \varphi$, and thus $\mu'_j \models_{\mathcal{D}} \varphi$. Thus, as in the proof of Theorem 1, $\mathcal{I} \models_{\mathcal{D}} \varphi$.
- Only if: If $\mathcal{I} \models_{\mathcal{D}} \varphi$, then $\mathcal{I} \models_{\mathcal{D}} \bigvee_j \mu_j$, and thus $\mathcal{I} \models_{\mathcal{D}} \mu_j$, for some j. □

As an immediate consequence of Lemma 2, we have the following theorem.

Theorem 2. *Let φ be a $\mathcal{D}$-formula and let $\mathcal{M} = \{\mu_1, \ldots, \mu_n\}$ be a complete collection of truth assignments propositionally $\mathcal{D}$-satisfying φ. Then, φ is $\mathcal{D}$-satisfiable if and only if μ_j is $\mathcal{D}$-satisfiable for some $\mu_j \in \mathcal{M}$.*

It follows from Theorem 2 that MATH-SAT is correct and complete if MATHSERVICE_PROVIDER is correct and complete. Notice that, unlike with Theorem 1, it is not necessary to check the whole set of total truth assignments $\mathcal{D}$-satisfying φ, rather it is sufficient to check a whatever complete collection $\mathcal{M}$ of partial assignments propositionally $\mathcal{D}$-satisfying φ, which is typically much smaller.

4.2 Efficiency issues

Apart for the efficiency of MATHSERVICE_PROVIDER —which varies with the kind of problem addressed and with the technique adopted, and will not be discussed here— a critical point for the efficiency of MATH-SAT is that the number of assignments in a complete set for φ is exponential in worst-case, and typically very big on average. This induces two considerations.

First, we don't want the procedure to require exponential space. Therefore we adopt a generate-check-and-drop paradigm: at each step, generate the next assignment $\mu_i \in \mathcal{M}$, check its $\mathcal{D}$-satisfiability, and then drop it —or drop the part of it which is not common to the next assignment— before passing to the $i+1$-step. This means that ASSIGN_ENUMERATOR must be able to generate the assignments one at a time. Thus, we avoid using techniques like, e.g., reduction to disjunctive normal form or Ordered Binary Decision Diagrams [5].

Second, we want to reduce as much as possible the number of assignments generated and checked. To do this, a key issue is choosing a procedure for ASSIGN_ENUMERATOR which generates non-redundant complete sets of assignments.

Definition 7. *We say that a complete collection $\mathcal{M} = \{\mu_1, \ldots, \mu_n\}$ of assignments propositionally $\mathcal{D}$-satisfying φ is* **non-redundant** *if for every $\mu_j \in \mathcal{M}$, $\mathcal{M}/\{\mu_j\}$ is no more complete,* **redundant** *otherwise.*

Definition 8. *We say that a complete collection $\mathcal{M} = \{\mu_1, \ldots, \mu_n\}$ of assignments propositionally $\mathcal{D}$-satisfying φ is* **strongly non-redundant** *if, for every $\mu_i, \mu_j \in \mathcal{M}$, $(\mu_1 \wedge \mu_2)$ is propositionally $\mathcal{D}$-unsatisfiable.*

Consequently, we say that ASSIGN_ENUMERATOR is

- *strongly non-redundant* if ASSIGN_ENUMERATOR(φ) is strongly non-redundant for every φ,
- *non-redundant* if ASSIGN_ENUMERATOR(φ) is non-redundant for every φ,
- *redundant* otherwise.

Example 4. Let $\varphi = (v_1 \leq v_2) \vee (v_3 \leq v_4)$. Then:

- $\{ \{(v_1 \leq v_2), (v_3 \leq v_4)\}, \{(v_1 \leq v_2), \neg(v_3 \leq v_4)\}, \{\neg(v_1 \leq v_2), (v_3 \leq v_4)\}\}$ is the set of all total assignments propositionally $\mathcal{D}$-satisfying φ;
- $\{ \{(v_1 \leq v_2)\}, \{(v_3 \leq v_4)\}, \{(v_1 \leq v_2), (v_3 \leq v_4)\} \}$ is complete but redundant, as the last assignment can be dropped without loosing completeness;
- $\{ \{(v_1 \leq v_2)\}, \{(v_3 \leq v_4)\}, \}$ is complete, non-redundant but not strongly redundant. In fact the two assignment are not mutually propositionally inconsistent, as both subsume the assignment $\{(v_1 \leq v_2), (v_3 \leq v_4)\}$;
- $\{ \{(v_1 \leq v_2)\}, \{\neg(v_1 \leq v_2), (v_3 \leq v_4)\}, \}$ is strongly non-redundant, as the two assignments are mutually inconsistent. $\diamond$

It is easy to verify that, if $\mathcal{M}$ is strongly non-redundant, then it is non-redundant too, but the vice versa does not hold.

If $\mathcal{M}$ is redundant, then $\mu_j \models_p \bigvee_{i \neq j} \mu_i$ for some j. Thus non-redundant assignment generators avoid generating partial assignments whose $\mathcal{D}$-unsatisfiability is a propositional consequence of that of already-generated ones. If $\mathcal{M}$ is strongly non-redundant, each total assignment η propositionally $\mathcal{D}$-satisfying φ is represented by one and only one $\mu_j \in \mathcal{M}$, and every $\mu_j \in \mathcal{M}$ represents univocally $2^{|Atoms(\varphi)|-|\mu_j|}$ total assignments. Thus strongly non-redundant assignment generators also avoid generating partial assignments covering areas of the search space which are covered by already-generated ones.

5 DPLL-based Algorithm(s)

To exploit the maximum efficiency we modify the standard DPLL algorithm so that to make it work as a strongly non-redundant truth assignment enumerator.

5.1 The basic DPLL-based algorithm

The basic non-CNF [3] DPLL-based procedure of MATH-SAT is reported in Figure 2. MATH-SAT takes in input a $\mathcal{D}$-formula φ and returns a truth value asserting whether φ is $\mathcal{D}$-satisfiable or not. MATH-SAT invokes MATH-DPLL passing as arguments φ and (by reference) an empty assignment μ and a null $\mathcal{D}$-interpretation $\mathcal{I}$. MATH-DPLL tries to build a $\mathcal{D}$-satisfiable truth assignment μ satisfying φ. This is done recursively, according to the following steps:

- (base) If $\varphi = T$, then μ propositionally $\mathcal{D}$-satisfies φ. Thus, if μ is $\mathcal{D}$-satisfiable, then φ is $\mathcal{D}$-satisfiable. Therefore MATH-DPLL invokes MATHSERVICE_PROVIDER(μ), which returns a $\mathcal{D}$-interpretation for μ if it is $\mathcal{D}$-satisfiable, $Null$ otherwise. MATH-DPLL returns $True$ in the first case, $False$ otherwise.

[3] To see how to use a DPLL with non-CNF formulas, see [2, 9]. In particular, [2] (and, indirectly, [9]) provide a notion of unit clause for non-CNF propositional formulas.

```
boolean MATH-SAT(formula φ)
    μ := {};
    I := Null;
    return MATH-DPLL(φ, μ, I);

boolean MATH-DPLL(formula φ, assignment & μ, interpretation & I)
    if (φ = T)                                              /* base      */
    then I := MATHSERVICE_PROVIDER(μ) ;
         return (I ≠ Null) ;
    if (φ = F)                                              /* backtrack */
    then return False;
    if {l occurs in φ as a unit clause}                     /* unit      */
    then return MATH-DPLL(assign(l, φ), μ ∪ {l}, I);
    l := choose-literal(φ);                                 /* split     */
    return   MATH-DPLL(assign(l, φ), μ ∪ {l}, I)   or
             MATH-DPLL(assign(¬l, φ), μ ∪ {¬l}, I);
```

Fig. 2. The basic version of MATH-SAT algorithm.

- (backtrack) If $\varphi = F$, then μ has lead to a propositional contradiction. Therefore MATH-DPLL returns $False$.
- (unit) If a literal l occurs in φ as a unit clause, then l must be assigned T.To obtain this, MATH-DPLL is invoked recursively with arguments the formula returned by *assign*(l, φ) and the assignment obtained by adding l to μ. *assign*(l, φ) substitutes every occurrence of l in φ with T and propositionally simplifies the result.
- (split) If none of the above situations occurs, then *choose-literal*(φ) returns an unassigned literal l according to some heuristic criterion. Then MATH-DPLL is first invoked recursively with arguments *assign*(l, φ) and $\mu \cup \{l\}$. If the result is $False$, then MATH-DPLL is invoked with arguments *assign*$(\neg l, \varphi)$ and $\mu \cup \{\neg l\}$.

MATH-DPLL is a variant of (a non-CNF version of) DPLL, modified to work as an enumerator of truth assignments, whose $\mathcal{D}$-satisfiability is recursively checked by MATHSERVICE_PROVIDER, in accordance with the generate-check-and-drop paradigm. The key difference with standard DPLL is in the "base" step: standard DPLL simply returns $True$, as it needs finding only one satisfying assignment.

5.2 Termination, correctness and completeness

At every recursive call, MATH-DPLL eliminates one of the finitely many atoms in $Atoms(\varphi)$. Thus, if every call to MATHSERVICE_PROVIDER terminates, then also MATH-SAT terminates.

To prove the correctness and completeness of MATH-SAT, we need the following result.

Theorem 3. *Given a $\mathcal{D}$-formula φ, the set of assignments generated and checked by the call* MATH-SAT(φ) *is complete and strongly non-redundant.*

Proof. Let μ^* be a possibly empty assignment and let $\mathcal{I}$ be $Null$. First, we prove that the set $\mathcal{M} = \{\mu_1, ..., \mu_n\}$ of assignments generated and checked by the call MATH-DPLL$(\varphi, \mu^*, \mathcal{I})$

is such that

$$\models_p (\varphi \wedge \mu^*) \leftrightarrow \bigvee_j \mu_j. \tag{5}$$

We reason by induction on the number of propositions occurring in φ.

- If $\varphi = True$, then (5) holds trivially as $n = 1$ and $\mu_1 = \mu^*$.
- If $\varphi = False$, then (5) holds trivially as $n = 0$.
- If l occurs in φ as a unit clause, then $\mathcal{M} = \mathcal{M}'$, $\mathcal{M}' = \{\mu_1, ..., \mu_l\}$ being the sets of assignments generated and checked by the call MATH-DPLL$(assign(\varphi, l), \mu^* \cup \{l\}, \mathcal{I})$. By inductive hypothesis,
$$\models_p (\varphi \wedge (\mu^* \wedge l)) \leftrightarrow \bigvee_j \mu_j.$$
As l occurs as a unit clause in φ, then $\models_p (\varphi \wedge (\mu^* \wedge l)) \leftrightarrow (\varphi \wedge \mu^*)$, so that (5) holds.
- In the general case, we have that $\mathcal{M} = \mathcal{M}' \cup \mathcal{M}''$, $\mathcal{M}' = \{\mu'_1, ..., \mu'_l\}$ and $\mathcal{M}'' = \{\mu''_1, ..., \mu''_m\}$ being the sets of assignments generated and checked respectively by the calls MATH-DPLL$(assign(\varphi, l), \mu^* \cup \{l\}, \mathcal{I})$ and MATH-DPLL$(assign(\varphi, \neg l), \mu^* \cup \{\neg l\}, \mathcal{I})$. By inductive hypothesis,
$$\models_p (\varphi \wedge (\mu^* \wedge l)) \leftrightarrow \bigvee_j \mu'_j \text{ and } \models_p (\varphi \wedge (\mu^* \wedge \neg l)) \leftrightarrow \bigvee_j \mu''_j,$$
from which we have that
$$\models_p (\varphi \wedge \mu^*) \leftrightarrow (\bigvee_j \mu'_j \vee \bigvee_j \mu''_j).$$

From (5) with $\mu^* = \{\}$, it follows that the set of assignments generated and checked by the call MATH-SAT(φ) is a complete set of assignments for φ.

Finally, it is straightforward to notice that, for every μ_i and μ_j generated and checked by the call MATH-DPLL$(\varphi, \mu, \mathcal{I})$, there is at least one literal l s.t. $l \in \mu_i$ and $\neg l \in \mu_j$, so that $\mu_i \wedge \mu_j$ is propositionally $\mathcal{D}$-unsatisfiable. Thus the set of assignments generated and checked by the call MATH-SAT(φ) is strongly non-redundant. □

It follows from Theorems 2 and 3 that MATH-SAT is correct and complete if MATHSERVICE_PROVIDER is correct and complete.

Notice that Theorem 3 does not hold for all SAT solvers. For instance, the sets of assignments generated by DPLL with *pure literal rule* [6] may be incomplete; the sets generated by standard analytic tableaux are complete but may be redundant; as for Ordered Binary Decision Diagrams (OBDDs) [5], all the sets of the paths leading to "1" are complete and strongly non-redundant, but they cannot be generated with a generate-check-and-drop policy.

5.3 Improvements & optimizations for MATH-DPLL

We describe some possible improvements and optimizations for MATH-DPLL, which come from adapting to our domain improvements and optimizations of the DPLL-based procedures for modal logics [10, 13, 12] and for temporal reasoning and resource planning [1, 18], which have proved extremely effective in their respective domains.

Preprocessing atoms [10, 13]. One potential source of inefficiency for the procedures of Figures 1 and 2 is the fact that semantically equivalent but syntactically different atoms are not recognized to be identical [resp. one the negation of the other] and thus they may be assigned different [resp. identical] truth values. For instance, syntactically different atoms may be equivalent modulo ordering ($(v_1 + v_2 \leq v_3)$, $(v_2 + v_1 \leq v_3)$), associativity ($(v_1 + (v_2 + v_3))$ and $((v_1 + v_2) + v_3)$)), factorization ($(2v_1 - 6v_2 = 14)$ and $(3v_1 - 9v_2 = 21)$), simple arithmetical term rewriting ($(v_1 + v_2 \leq v_3)$ and $(v_1 + v_2 - v_3 \leq 0)$) and so on, or may be one the negation of the other ($(v_1 < v_2)$ and $(v_1 \geq v_2)$, or $(2v_1 - 6v_2 \leq 4)$ and $(3v_2 + 2 < v_1)$). This causes the undesired generation of a potentially very big amount of intrinsically $\mathcal{D}$-unsatisfiable assignments (for instance, up to $2^{Atoms(\varphi)-2}$ assignments of the kind $\{(v_1 < v_2), (v_1 \geq v_2), ...\}$).

To avoid these problems, it is wise to preprocess atoms so that to map semantically equivalent atoms into syntactically identical ones. Of course, this mapping depends on the problem addressed (integers/reals, linear/nonlinear, equalities/inequalities, etc). Some common steps can be:

- *exploit associativity* (e.g., $(v_1 + (v_2 + v_3))$ and $((v_1 + v_2) + v_3)) \Longrightarrow (v_1 + v_2 + v_3)$);
- *sorting* (e.g., $(v_1 + v_2 \leq v_3 + 1)$, $(v_2 + v_1 \leq v_3 + 1)$, $(v_1 + v_2 - 1 \leq v_3) \Longrightarrow (v_1 + v_2 - v_3 \leq 1)$);
- *factoring* (e.g., $(v_1 + 2.0v_2 - 3.0v_3 \leq 4.0)$, $(-2.0v_1 - 4.0v_2 + 6.0v_3 \geq -8.0)$, $\Longrightarrow$ $(0.25v_1 + 0.5v_2 - 0.75v_3 \leq 1.0)$);
- *exploiting negation* (e.g., $(v_1 < v_2)$, $(v_1 \geq v_2) \Longrightarrow (v_1 < v_2)$, $\neg(v_1 < v_2)$);
- *exploiting integer properties* (e.g., $(v_1 \leq 3)$, $(v_1 < 4) \Longrightarrow (v_1 \leq 3)$);
- ...

Intermediate Assignment Checking [10]. Another improvement starts from the observation that typically most assignments found by MATH-DPLL are "trivially" $\mathcal{D}$-unsatisfiable, in the sense that their $\mathcal{D}$-unsatisfiability is caused by much smaller subsets, which we call *conflict sets*. If an assignment μ' is $\mathcal{D}$-unsatisfiable, then all its extensions are $\mathcal{D}$-unsatisfiable. If the unsatisfiability of an assignment μ' is detected during its recursive construction, then this prevents checking the $\mathcal{D}$-satisfiability of all the up to $2^{|Atoms(\varphi)|-|\mu'|}$ truth assignments which extend μ'.

This suggests to introduce an intermediate $\mathcal{D}$-satisfiability test on incomplete assignments just before the "split" step:

---

if *Likely-Unsatisfiable*(μ) /* intermediate assignment check */
  if (MATHSERVICE_PROVIDER(μ) = $Null$)
    then return *False;*

If the heuristic *Likely-Unsatisfiable* returns $True$, then MATHSERVICE_PROVIDER is invoked on the current assignment μ. If MATHSERVICE_PROVIDER(μ) returns $Null$, then all possible extensions of μ are unsatisfiable, and therefore MATH-DPLL returns $False$, avoiding a possibly big amount of useless search.

Example 5. Consider the formula φ of Example 1. Suppose that, after four recursive calls, MATH-DPLL builds the intermediate assignment:

$$\mu' = \neg(2v_2 - v_3 > 2) \wedge \neg A_2 \wedge (3v_1 - 2v_2 \leq 3) \wedge \neg(3v_1 - v_3 \leq 6). \quad (6)$$

(rows 1, 2, 3 and 4 of φ). If MATHSERVICE_PROVIDER is invoked on μ', it returns $Null$, and MATH-DPLL backtracks without exploring any extension of μ'. $\diamond$

Likely-Unsatisfiable avoids invoking MATHSERVICE_PROVIDER when it is very unlikely that, since last call, the new literals added to μ' can cause inconsistency. For instance, this is the case when they are added only literals which either are purely-propositional or contain new variables.

In theory, the introduction of an intermediate consistency check before *every* split could negatively affect the global worst-case performance. In practice, it is sufficient an average pruning of one single split per branch to make the intermediate consistency check worth doing.

Backjumping [13, 18]. An alternative optimization starts from the same observations as those of Intermediate Assignment Checking. Any branch containing a conflict set is $\mathcal{D}$-unsatisfiable. Thus, suppose MATHSERVICE_PROVIDER is modified to return also a conflict set η causing the $\mathcal{D}$-unsatisfiability of the input assignment μ. If so, MATH-DPLL can jump back in its search to the deepest branching point in which a literal $l \in \eta$ is assigned a truth value, pruning the search space below.

Example 6. Consider the formula φ and the assignment μ of Example 1. Suppose that MATH-DPLL generates μ following the order of occurrence within φ, and that MATHSERVICE_PROVIDER(μ) returns the conflict set (3). Thus MATH-DPLL can backjump directly to the branching point $\neg(3v_1 - v_3 \leq 6)$ without branching first on $(v_3 = 3v_5 + 4)$ and $\neg(2v_2 - v_3 > 2)$. $\diamond$

Learning [18, 1]. When MATHSERVICE_PROVIDER returns a conflict set η, the clause $\neg\eta$ can be added in conjunction to φ. Since then, MATH-DPLL will never again generate any branch containing η.

Example 7. As in Example 6, suppose MATHSERVICE_PROVIDER(μ) returns the conflict set (3). Then the clause

$$\neg(3v_1 - 2v_2 \leq 3) \vee (2v_2 - v_3 > 2) \vee (3v_1 - v_3 \leq 6)$$

is added in conjunction to φ. Thus, whenever a branch contains two elements of (3), then MATH-DPLL will assign the third to $False$ by unit propagation. $\diamond$

Learning is a technique which must be used with some care, as it may cause an explosion in size of φ. To avoid this, one has to introduce techniques for discarding learned clauses when necessary [3].

Strictly related to learning is a preprocessing technique described in [1]. Before starting the search, for a given (small) n (e.g., $n = 2, 3$), for all the possible clauses C given by the combination of up to n non-elementary $\mathcal{D}$-literals of φ, MATHSERVICE_PROVIDER is run on C and, if it returns $Null$, C is added to φ. Thus, MATH-DPLL is forced to learn a priori all the possible incompatibility constraints of length up to n. If $n = 2$, this preprocess step requires $O(|Atoms(\varphi)|^2)$ steps.

Triggering This technique is a generalization we propose of a technique adopted in [18], and comes from the following consideration.

Property 5. Let φ be a $\mathcal{D}$-formula and let C be a non-elementary $\mathcal{D}$-atom occurring only positively [resp. negatively] in φ. Let $\mathcal{M}$ be a complete set of assignments for φ, and let

$$\mathcal{M}' := \{\mu_j/\neg C\}|\mu_j \in \mathcal{M}\} \quad [resp.\ \{\mu_j/C\}|\mu_j \in \mathcal{M}\}].$$

Then φ is $\mathcal{D}$-satisfiable if and only if there exist a $\mathcal{D}$-satisfiable $\eta' \in \mathcal{M}'$ s.t. $\eta' \models_p \varphi$.

Proof. We prove the positive case, as the negative is dual. The "if" case is trivial. If φ is $\mathcal{D}$-satisfiable, then there is a $\mathcal{D}$-satisfiable $\eta \in \mathcal{M}$ s.t. $\eta \models_p \varphi$ because $\mathcal{M}$ is complete. If $\neg C \notin \eta$, then we the thesis holds with $\eta' := \eta$. If $\neg C \in \eta$, then let $\eta' := \eta/\neg C$. η' is trivially $\mathcal{D}$-satisfiable. As $\eta \models_p \varphi$ and C occurs only positively in φ, $\eta' \models_p \varphi$

Thus, if we have non-elementary $\mathcal{D}$-atoms in the input formulas, we can drop any negative occurrence of them from the assignment to be checked by MATHSERVICE_PROVIDER. This is particularly useful when we deal with equality atoms like, e.g., $(v_1 - v_2 = 3.2)$, as handling checking negative equalities like $(v_1 - v_2 \neq 3.2)$ forces splitting it into the disjunction of two strict inequalities $(v_1 - v_2 > 3.2) \vee (v_1 - v_2 < 3.2)$.

6 Related work

This work extends our previous work on decision procedures for modal and description logics based on SAT procedures. [10, 11, 7] presented the first such procedures for the modal logic K, $K(m)$ and the description logic $\mathcal{ALC}$, which outperformed both standard tableau based and translation based procedures; [16, 12] provided a general schema for SAT-based procedures for every normal modal logic, and a general formal framework to build SAT based procedures; [8] presented SAT based procedures for non-normal modal logics.

[1] presented an optimized SAT based procedure for sets of disjunctive temporal constraints which outperformed previous tableau-based procedures for the same problem; [18] presented a SAT based procedure for sets of disjunctive equalities and non-strict inequalities, and used it to solve very efficiently problems in the domain of resource planning. Both procedures integrate DPLL with a simplex LP tool, and accept only non-negated atomic constraints.

In the Model Checking community there has been some work on integrating OBDDs [5] with numerical information in order to handle complex verification problems. [17] integrated OBDDs with a quadratic constraint solver to verify transition systems with integer data values; [14] developed Difference Decision Diagrams (DDDs), OBDD-like data structures handling boolean combinations of temporal constraints, and used them to verify timed systems; [19] developed a library of procedures combining OBDDs and a solver for Pressburger arithmetic, and used them to verify infinite-state systems. Unfortunately, all these approaches inherit from OBDDs the drawback of requiring exponential space in worst case.

References

1. A. Armando, C. Castellini, and E. Giunchiglia. SAT-based procedures for temporal reasoning. In *Proc. European Conference on Planning, CP-99*, 1999.
2. A. Armando and E. Giunchiglia. Embedding Complex Decision Procedures inside an Interactive Theorem Prover. *Annals of Mathematics and Artificial Intelligence*, 8(3–4):475–502, 1993.
3. R. J. Bayardo, Jr. and R. C. Schrag. Using CSP Look-Back Techniques to Solve Real-World SAT instances. In *Proc AAAI'97*, pages 203–208. AAAI Press, 1997.
4. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. CAV'99*, 1999.
5. R. E. Bryant. Symbolic Boolean manipolation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.
6. M. Davis, G. Longemann, and D. Loveland. A machine program for theorem proving. *Journal of the ACM*, 5(7), 1962.
7. E. Giunchiglia, F. Giunchiglia, R. Sebastiani, and A. Tacchella. More evaluation of decision procedures for modal logics. In *Proc. Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, Trento, Italy, 1998.
8. E. Giunchiglia, F. Giunchiglia, and A. Tacchella. SAT-Based Decision Procedures for Classical Modal Logics. *Journal of Automated Reasoning*, 2000. To appear.
9. E. Giunchiglia and R. Sebastiani. Applying the Davis-Putnam procedure to non-clausal formulas. In *Proc. AI*IA'99*, number 1792 in Lecture Notes in Artificial Intelligence. Springer Verlag, 1999.
10. F. Giunchiglia and R. Sebastiani. Building decision procedures for modal logics from propositional decision procedures - the case study of modal K. In *Proc. of the 13th Conference on Automated Deduction*, LNAI, New Brunswick, NJ, USA, August 1996. Springer Verlag.
11. F. Giunchiglia and R. Sebastiani. A SAT-based decision procedure for ALC. In *Proc. of the 5th International Conference on Principles of Knowledge Representation and Reasoning - KR'96*, Cambridge, MA, USA, November 1996.
12. F. Giunchiglia and R. Sebastiani. Building decision procedures for modal logics from propositional decision procedures - the case study of modal K(m). *Information and Computation*, 162(1/2), October/November 2000.
13. I. Horrocks and P. F. Patel-Schneider. FaCT and DLP. In *Proceedings of International Conference on Automated Reasoning with Analytic Tableaux and Related Methods – Tableaux'98*, number 1397 in Lecture Notes in Artificial Intelligence, pages 27–30. Springer-Verlag, 1998.
14. Henrik R. Andersen Jesper Moeller, Jakob Lichtenberg and Henrik Hulgaard. Fully symbolic model checking of timed systems using difference decision diagrams. In *Proc. Workshop on Symbolic Model Checking (SMC), Federated Logic Conference (FLoC)*, Trento, Italy, July 1999.
15. H. Kautz, D. McAllester, and Bart Selman. Encoding Plans in Propositional Logic. In *Proc. KR'96*, 1996.
16. Roberto Sebastiani and Adolfo Villafiorita. SAT-based decision procedures for normal modal logics: a theoretical framework. In *Proc. 6th International Conference on Artificial Intelligence: Methodology, Systems, Applications - AIMSA'98*, number 1480 in LNAI, Sozopol, Bulgaria, September 1998. Springer Verlag.
17. P. Beame W. Chan, R. J. Anderson and D. Notkin. Combining constraint solving and symbolic model checking for a class of systems with non-linear constraints. In *Proc. CAV'97*, volume 1254 of *LNCS*, pages 316–327, Haifa, Israel, June 1997. Springer-Verlag.
18. S. Wolfman and D. Weld. The LPSAT Engine & its Application to REsource Planning. In *Proc. IJCAI*, 1999.
19. T. Yavuz-Kahveci, M. Tuncer, and T. Bultan. A Library for Composite Symbolic Representation. In *Proc. TACAS2001, Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *LNCS*. Springer Verlag, 2000.