Integrating Boolean and Mathematical Solving: Foundations, Basic Algorithms and Requirements ⋆

Gilles Audemard[1 2], Piergiorgio Bertoli[1], Alessandro Cimatti[1], Artur Korniłowicz[1 3], and Roberto Sebastiani[1 4]

[1] ITC-IRST, Povo, Trento, Italy
{audemard,bertoli,cimatti,kornilow}@itc.it
[2] LSIS, University of Provence, Marseille, France
[3] Institute of Computer Science, University of Białystok, Poland
[4] DIT, Università di Trento, Povo, Trento, Italy
roberto.sebastiani@dit.unitn.it

Abstract. In the last years we have witnessed an impressive advance in the efficiency of boolean solving 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 solvers, 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 boolean and mathematical solvers, and we highlight the main requirements that boolean and mathematical solvers must fulfill in order to achieve the maximum benefits from their integration. Finally we show how existing systems are captured by our framework.

1 Motivation and goals

In the last years we have witnessed an impressive advance in the efficiency of boolean solving techniques (SAT), 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.

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, 15, 5, 10]). 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., [5, 10]). On the other hand, mathematical solvers, like

⋆ This work is sponsored by the CALCULEMUS! IHP-RTN EC project, contract code HPRN-CT-2000-00102, and has thus benefited of the financial contribution of the Commission through the IHP programme.

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 for building domain-specific decision procedures on top of SAT solvers [7, 8]. 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 [8], temporal reasoning [1], resource planning [15].

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 integrate SAT and mathematical solvers, and we highlight the main requirement SAT and mathematical solvers must fulfil in order to achieve the maximum benefits from their integration. The ultimate goal is to develop solvers able to handle complex 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 which combine boolean and mathematical solvers and introduce some efficiency issues. In Section 5 we highlight the main requirements that boolean and mathematical solvers must fulfil in order to achieve the maximum benefits from their integration. In Section 6 we briefly describe some existing systems which are captured by our framework,and our own implementated procedure.

For lack of space, in this paper we omit the proofs of all the theoretical results presented, which can be found in [13].

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 *Math-terms* the mathematical expressions built on constants, variables and arithmetical operators over $\mathcal{D}$: a constant $c_i \in \mathcal{C}$ is a Math-term; a variable $v_i \in \mathcal{V}$ is a Math-term; if t_1 is a Math-term, then $-t_1$ is a Math-term; if t_1, t_2 are Math-terms, then $(t_1 \otimes t_2)$ is a Math-term, $\otimes \in \mathcal{OP}_{\mathcal{D}}$.

We call *Math-formulas* the mathematical formulas built on primitive propositions, Math-terms, operators and boolean connectives: a primitive proposition $A_i \in \mathcal{A}$ is a Math-formula; if t_1, t_2 are Math-terms, then $(t_1 \bowtie t_2)$ is a Math-formula, $\bowtie \in \{=, \neq, >, <, \geq, \leq\}$; if φ_1 is a Math-formula, then $\neg\varphi_1$ is a Math-formula; if φ_1, φ_2 are Math-formulas, then $(\varphi_1 \wedge \varphi_2)$ is a Math-formula. For instance, $A_1 \wedge ((v_1 + 5.0) \leq (2.0 \cdot v_3))$ and $A_2 \wedge \neg(((2 \cdot v_1)\ mod\ v_2) > 5)$ are Math-formulas.

Notationally, we use the downcase letters $t, t_1, ...$ to denote Math-terms, and the Greek letters $\alpha, \beta, \varphi, \psi$ to denote Math-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 "$\neg(\varphi_1 \wedge \neg\varphi_2) \wedge \neg(\varphi_2 \wedge \neg\varphi_1)$", "$\top$" for any valid formula, and "$\bot$" for "$\neg\top$". When this does not cause ambiguities, we use the associativity and precedence rules of arithmetical operators to simplify the appearance of Math-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 *interpretation* a map $\mathcal{I}$ which assigns $\mathcal{D}$ values and boolean values to Math-terms and Math-formulas respectively and preserves constants and arithmetical operators:[1]

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

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

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

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

3 The formal framework

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

Examples of literals are, A_1, $\neg A_2$, $(v_1 + 5.0 \leq 2.0 v_3)$, $\neg((2v_1 \ mod\ v_2) > 5)$. 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.

Definition 2. *We call a* **total truth assignment** μ *for a Math-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*.*

[1] 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 $\{=, \neq, >, <, \geq, \leq\}$.

A total truth assignment μ like (1) 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. We often write a truth assignment μ as the conjunction of its elements. To this extent, we say that μ is satisfiable if the conjunction of its elements is satisfiable.

Definition 3. *We say that a total truth assignment μ for φ* **propositionally 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:*

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

We say that a partial truth assignment μ **propositionally satisfies** *φ if and only if all the total truth assignments for φ which extend μ propositionally satisfy φ.*

We say that φ is *propositionally satisfiable* if and only if there exist an assignment μ s.t. $\mu \models_p \varphi$. Intuitively, if we consider a Math-formula φ as a propositional formulas in its atoms, then $\models_p$ is the standard satisfiability in propositional logic. All the standard properties of $\models_p$ hold consequently. 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 unsatisfiable.

Notice that $\models_p$ is stronger than $\models$, that is, if $\varphi_1 \models_p \varphi_2$, then $\varphi_1 \models \varphi_2$, but not vice versa. E.g., $(v_1 \leq v_2) \wedge (v_2 \leq v_3) \models (v_1 \leq v_3)$, 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 math-formula φ:

$$\begin{array}{rl} \varphi = & \{\underline{\neg(2v_2 - v_3 > 2)} \vee A_1\} \wedge \\ & \{\underline{\neg A_2} \vee (2v_1 - 4v_5 > 3)\} \wedge \\ & \{\underline{(3v_1 - 2v_2 \leq 3)} \vee A_2\} \wedge \\ & \{\neg(2v_3 + v_4 \geq 5) \vee \underline{\neg(3v_1 - v_3 \leq 6)} \vee \neg A_1\} \wedge \\ & \{A_1 \vee \underline{(3v_1 - 2v_2 \leq 3)}\} \wedge \\ & \{\underline{(v_1 - v_5 \leq 1)} \vee (v_5 = 5 - 3v_4) \vee \neg A_1\} \wedge \\ & \{A_1 \vee \underline{(v_3 = 3v_5 + 4)} \vee A_2\}. \end{array}$$

The truth assignment given by the underlined literals above is:

$$\mu = \{\neg(2v_2 - v_3 > 2), \neg A_2, (3v_1 - 2v_2 \leq 3), (v_1 - v_5 \leq 1), \neg(3v_1 - v_3 \leq 6), (v_3 = 3v_5 + 4)\}.$$

μ is an assignment which propositionally satisfies φ, as it sets to true one literal of every disjunction in φ. Notice that μ is not satisfiable, as both the following sub-assignments of μ

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

$$\{(v_1 - v_5 \leq 1), (v_3 = 3v_5 + 4), \neg(3v_1 - v_3 \leq 6)\} \tag{3}$$

do not have any satisfying interpretation. $\diamond$

Definition 4. *We say that a collection $\mathcal{M} = \{\mu_1, \ldots, \mu_n\}$ of (possibly partial) assignments propositionally 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 satisfying φ. Notice however that $||\mathcal{M}||$ is worst-case exponential in the size of φ, though typically much smaller than the set of all total assignments satisfying φ.

Definition 5. *We say that a complete collection $\mathcal{M} = \{\mu_1, \ldots, \mu_n\}$ of assignments propositionally satisfying φ is* **non-redundant** *if for every $\mu_j \in \mathcal{M}$, $\mathcal{M}/\{\mu_j\}$ is no more complete,* **redundant** *otherwise. $\mathcal{M}$ is* **strongly non-redundant** *if, for every $\mu_i, \mu_j \in \mathcal{M}$, $(\mu_1 \wedge \mu_2)$ is propositionally unsatisfiable.*

It is easy to verify that, if $\mathcal{M}$ is redundant, then $\mu_i \subseteq \mu_j$ for some i, j, and that, if $\mathcal{M}$ is strongly non-redundant, then it is non-redundant too, but the vice versa does not hold.

Example 2. Let $\varphi := (\alpha \vee \beta \vee \gamma) \wedge (\alpha \vee \beta \vee \neg\gamma)$, α, β and γ being atoms. Then

1. $\{\{\alpha, \beta, \gamma\}, \{\alpha, \beta, \neg\gamma\}, \{\alpha, \neg\beta, \gamma\}, \{\alpha, \neg\beta, \neg\gamma\}, \{\neg\alpha, \beta, \gamma\}, \{\neg\alpha, \beta, \neg\gamma\}\}$ is the set of all total assignments propositionally satisfying φ;
2. $\{\{\alpha\}, \{\alpha, \beta\}, \{\alpha, \neg\gamma\}, \{\alpha, \beta\}, \{\beta\}, \{\beta, \neg\gamma\}, \{\alpha, \gamma\}, \{\beta, \gamma\}\}$ is complete but redundant;
3. $\{\{\alpha\}, \{\beta\}\}$ is complete, non redundant but not strongly non-redundant;
4. $\{\{\alpha\}, \{\neg\alpha, \beta\}\}$ is complete and strongly non-redundant. $\diamond$

Theorem 1. *Let φ be a Math-formula and let $\mathcal{M} = \{\mu_1, \ldots, \mu_n\}$ be a complete collection of truth assignments propositionally satisfying φ. Then φ is satisfiable if and only if μ_j is satisfiable for some $\mu_j \in \mathcal{M}$.*

Having a Math-formula φ, it is always possible to find a complete collection of satisfying assignments for φ (see later). Thus from Theorem 1 we have trivially the following fact.

Proposition 1. *The satisfiability problem for a Math-formula over atoms of a given class is decidable if and only if the satisfiability of sets of literals of the same class is decidable.*

For instance, the satisfiability of a set of linear constraints on $\mathbb{R}$ or on $\mathbb{Z}$, or a set of non-linear constraints on $\mathbb{R}$ is decidable, whilst a set of non-linear (polynomial) constraints on $\mathbb{Z}$ is not decidable (see, e.g., [12]). Consequently, the satisfiability of Math-formulas over linear constraints on $\mathbb{R}$ or on $\mathbb{Z}$, or over non-linear constraints on $\mathbb{R}$ is decidable, whilst the satisfiability of Math-formulas over non-linear constraints over $\mathbb{Z}$ is undecidable.

For the decidable cases, as standard boolean formulas are a strict subcase of Math-formulas, it follows trivially that deciding the satisfiability of Math-formulas is "at least as hard" as boolean satisfiability.

Proposition 2. *The problem of deciding the satisfiability of a Math-formula φ is NP-hard.*

Thus, deciding 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 arithmetical expressions over bounded integers, then for every $\mathcal{I}$ we can verify $\mathcal{I} \models \varphi$ in a polynomial amount of time, and thus the problem is also NP-complete.

```
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 := MATHSOLVER(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 satisfiability of a Math-formula φ into two orthogonal components: a *purely boolean* component, consisting in the existence of a propositional model for φ, and a *purely mathematical* component, consisting in the existence of a 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 solver.*

Definition 6. *We call a* **truth assignment enumerator** *a total function* ASSIGN_ENUMERATOR *which takes as input a Math-formula φ and returns a complete collection $\{\mu_1, \ldots, \mu_n\}$ of assignments satisfying φ.*

Notice the difference between a truth assignment enumerator and a standard boolean solver: a boolean solver has to find *only one* satisfying assignment —or to decide there is none— while an enumerator has to find a *complete collection* of satisfying assignments. (We will show later how some boolean solvers can be modified to be used as enumerators.)

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.

Definition 7. *We call a* **mathematical solver** *a total function* MATHSOLVER *which takes as input a set of (possibly negated) atomic Math-formulas μ and returns a interpretation satisfying μ, or Null if there is none.*

The general schema of a search procedure for 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 interpretation $\mathcal{I}$. For every assignment μ in the collection $\{\mu_1, .., \mu_n\}$ generated by ASSIGN_ENUMERATOR(φ), MATH-SAT invokes MATHSOLVER over μ, which either returns a interpretation satisfying μ, or $Null$ if there is none. This is done until either one satisfiable assignment is found, or no more assignments are available in $\{\mu_1, ..., \mu_n\}$. In the former case φ is satisfiable, in the later case it is not.

MATH-SAT performs at most $||\mathcal{M}||$ loops. Thus, if every call to MATHSOLVER terminates, then MATH-SAT terminates. Moreover, it follows from Theorem 1 that MATH-SAT is correct and complete if MATHSOLVER is correct and complete. Notice that, it is not necessary to check the whole set of total truth assignments satisfying φ, rather it is sufficient to check a whatever complete collection $\mathcal{M}$ of partial assignments propositionally satisfying φ, which is typically much smaller.

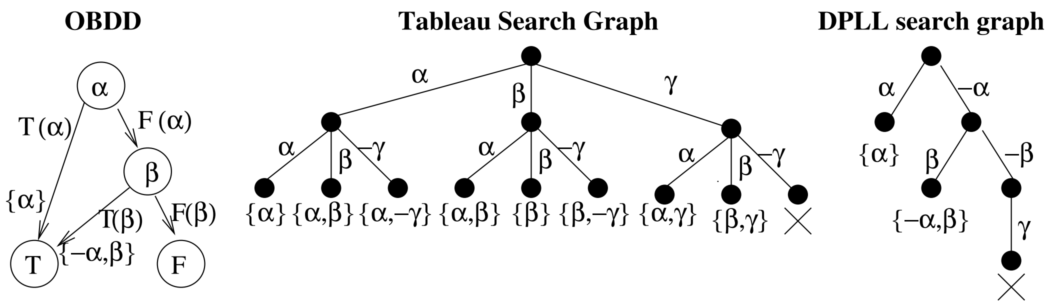

Fig. 2. Tableau search graph, DPLL search graph and OBDD for the formula $(\alpha \vee \beta \vee \gamma) \wedge (\alpha \vee \beta \vee \neg\gamma)$.

4.1 Suitable ASSIGN_ENUMERATORs

The following are the most significant boolean reasoning techniques that we can adapt to be used as assignment enumerators.

DNF. The simplest technique we can use as an enumerator is the *Disjunctive Normal Form (DNF)* conversion. A propositional formula φ can be converted into a formula $DNF(\varphi)$ by (i) recursively applying DeMorgan's rewriting rules to φ until the result is a disjunction of conjunction of literals, and (ii) removing all duplicated and subsumed disjuncts. The resulting formula is normal, in the sense that $DNF(\varphi)$ is propositionally equivalent to φ, and that propositionally equivalent formulas generate the same DNF modulo reordering.

By Definition 4, we can see (the set of disjuncts of) $DNF(\varphi)$ as a complete and non-redundant —but not strongly non-redundant— collection of assignments propositionally satisfying φ. For instance, in Example 2, the set of assignments at point 2. and 3. are respectively the results of step (i) and (ii) above.

OBDD. A more effective normal form for representing a boolean formula if given by the *Ordered Binary Decision Diagrams (OBDDs)* [4], which are extensively used in hardware verification and model checking. Given a total ordering $v_1, ..., v_n$ on the atoms of φ, the OBDD representing φ ($OBDD(\varphi)$) is a directed acyclic graph such that (i) each node is either one of the two terminal nodes T, F, or an internal node labeled by an atom v of φ, with two outcoming edges $T(v)$ ("v is true") and $F(v)$ ("v is false"), (ii) each arc $v_i \to v_j$ is such that $v_i < v_j$ in the total order. If a node n labeled with v is the root of $OBDD(\phi)$ and n_1, n_2 are the two son nodes of n through the edges $T(v)$ and $F(v)$ respectively, then n_1, n_2 are the roots of $OBDD(\phi[v = \top])$ and $OBDD(\phi[v = \bot])$ respectively. A path from the root of $OBDD(\varphi)$ to T [resp. F] is a propositional model [resp. counter-model] of φ, and the disjunction of such paths is propositionally equivalent to φ [resp. $\neg\varphi$].

Thus, we can see $OBDD(\varphi)$ as a complete collection of assignments propositionally satisfying φ. As every pair of paths differ for the truth value of at least one variable, $OBDD(\varphi)$ is also strongly non-redundant. For instance, in Figure 2 (left) it is represented the OBDD of the formula in Example 2. The paths to T are those given by the set of assignments at point 4. of Example 2.

Semantic tableaux. A standard boolean solving technique is that of semantic tableaux [14]. Given an input formula φ, in each branch of the search tree the set $\{\varphi\}$ is decomposed into a set of literals μ by the recursive application of the rules:

$$\frac{\mu' \cup \{\bigwedge_{i=1}^{n} \varphi_i\}}{\mu' \cup \{\varphi_1, ..., \varphi_n\}} \ (\wedge) \qquad \frac{\mu' \cup \{\bigvee_{i=1}^{n} \varphi_i\}}{\mu' \cup \{\varphi_1\} \qquad .... \qquad \mu' \cup \{\varphi_n\}} \ (\vee)$$

plus analogous rules for $(\rightarrow)$, $(\leftrightarrow)$, $(\neg\wedge)$, $(\neg\vee)$, $(\neg \rightarrow)$, $(\neg \leftrightarrow)$. The main steps are:

- (closed branch) if μ contains both φ_i and φ_i for some subformula φ_i of φ, then μ is said to be *closed* and cannot be decomposed any further;
- (solution branch) if μ contains only literals, then it is an assignment s.t. $\mu \models_p \varphi$;
- ($\wedge$-rule) if μ contains a conjunction, then the latter is unrolled into the set of its conjuncts;
- ($\vee$-rule) if μ contains a disjunction, then the search branches on one of the disjuncts.

The search tree resulting from the decomposition is such that all its solution branches are assignments in a collection $Tableau(\varphi)$, whose disjunction is propositionally equivalent to φ. Thus $Tableau(\varphi)$ is complete, but it may be redundant. For instance, in Figure 2 (center) it is represented the search tree of a semantic tableau applied on the formula in Example 2. The solutions branches give rise to the redundant collection of assignments at point 2. of Example 2.

DPLL. The most commonly used boolean solving procedure is DPLL [6]. Given φ in input, DPLL tries to build recursively one assignment μ satisfying φ, at each step adding a new literal to μ and simplifying φ, according to the following steps:

- (base) If $\varphi = \top$, then μ propositionally satisfies the original formula, so that μ is returned;
- (backtrack) if $\varphi = \bot$, μ propositionally falsifies the original formula, so that DPLL backtracks;
- (unit propagation) if a literal l occurs in φ as a unit clause, then DPLL is invoked recursively on $\varphi_{l=\top}$ and $\mu \cup \{l\}$, $\varphi_{l=\top}$ being the result of substituting $\top$ for l in φ and simplifying;
- (split) otherwise, a literal l is selected, and DPLL is invoked recursively on $\varphi_{l=\top}$ and $\mu \cup \{l\}$. If this call succeeds, then the result is returned, otherwise DPLL is invoked recursively on $\varphi_{l=\bot}$ and $\mu \cup \{\neg l\}$.

Standard DPLL can be adapted to work as a boolean enumerator by simply modifying in the base case "μ is returned" with "μ is added to the collection, and backtrack" [7, 13].

The resulting set of assignments $DPLL(\varphi)$ is complete and strongly non-redundant [13]. For instance, in Figure 2 (right) it is represented the search tree of DPLL applied on the formula in Example 2. The non closed branches give rise to the set of assignments at point 4. of Example 2.

Notice the difference between an OBDD and (the search tree of) DPLL: first, the former is a direct acyclic graph whilst the second is a tree; second, in OBDDs the order of branching variables if fixed a priori, while DPLL can choose each time the best variable to split.

4.2 Non-suitable ASSIGN_ENUMERATORs

It is very important to notice that, in general, not every boolean solver can be adapted to work as a boolean enumerator. For instance, many implementation of DPLL include also the following step between unit propagation and split:

- (pure literal) if an atom ψ occurs only positively [resp. negatively] in φ, then DPLL is invoked recursively on $\varphi_{\psi=\top}$ and $\mu \cup \{\psi\}$ [resp. $\varphi_{\psi=\bot}$ and $\mu \cup \{\neg\psi\}$];

(we call this variant DPLL+PL). DPLL+PL is complete as a boolean solver, but does not generate a complete collection of assignments, so that it cannot be used as an enumerator.

Example 3. If we used DPLL+PL as ASSIGN_ENUMERATOR in MATH-SAT and gave in input the formula in Example 2, DPLL+PL might return the one-element collection $\{\{\alpha\}\}$, which is not complete. If α is $(x^2 + 1 \leq 0)$ and β is $(y \leq x)$, $x, y \in \mathbb{R}$, then $\{\alpha\}$ is not satisfiable, so that MATH-SAT would return unsatisfiable. On the other hand, the formula φ is satisfiable because, e.g., the assignment $\{\neg\alpha, \beta\}$ is satisfied by $\mathcal{I}(x) = 1.0$ and $\mathcal{I}(y) = 0.0$.

5 Requirements for ASSIGN_ENUMERATOR and MATHSOLVER

Apart for the efficiency of MATHSOLVER —which varies with the kind of problem addressed and with the technique adopted, and will not be discussed here— and that of the SAT solver used —which does not necessarily imply its efficiency as an enumerator— many other factors influence the efficiency of MATH-SAT.

Polynomial vs. exponential space in ASSIGN_ENUMERATOR. We would rather MATH-SAT not to require exponential space. As $\mathcal{M}$ can be exponentially big w.r.t. the size of φ, we would rather adopt a generate-check-and-drop paradigm: at each step, generate the next assignment $\mu_i \in \mathcal{M}$, check its 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.

To this extent, both DNF and OBDD are not an suitable choice for ASSIGN_ENUMERATOR, as they force generating the whole assignment collection $\mathcal{M}$ one-shot. Both semantic tableaux and DPLL instead are a good choice, as their depth-first search strategy allows for generating and checking one assignment at a time.

Non-redundancy of ASSIGN_ENUMERATOR. We want to reduce as much as possible the number of assignments generated and checked. To do this, a key issue is avoiding MATHSOLVER being invoked on an assignment which either is identical to an already-checked one or extends one which has been already found unsatisfiable. This is obtained by using a non-redundant enumerator. To this extent, semantic tableaux are not a good choice.

Non-redundant enumerators avoid generating partial assignments whose unsatisfiability is a propositional consequence of that of already-generated ones. If $\mathcal{M}$ is strongly non-redundant, however, each total assignment η propositionally 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 enumerators also avoid generating partial assignments covering areas of the search space which are covered by already-generated ones.

For enumerators that are not strongly non-redundant, there is a tradeoff between redundancy and polynomial memory. In fact, when adopting a generate-check-and-drop paradigm, the algorithm has no way to remember if it has already checked a given assignment or not, unless it explicitly keeps track of it, which requires up to exponential memory. Strong non-redundancy instead provides a *logical* warrant that an already checked assignment will never be checked again.

Intermediate assignment checking. When the unsatisfiability of an assignment μ' is detected during its recursive construction, this prevents checking the satisfiability of all the up to $2^{|Atoms(\varphi)|-|\mu'|}$ truth assignments which extend μ'. Thus, another key issue for efficiency is the possibility of modifying ASSIGN_ENUMERATOR so that it can perform intermediate calls to MATHSOLVER and it can take advantage of the (un)satisfiability information returned to prune the search space.

With semantic tableaux and DPLL, this can be easily obtained by introducing an intermediate test, immediately before the ($\vee$-rule) and the (split) step respectively, in which MATHSOLVER is invoked on an intermediate assignment μ' and, if it is inconsistent, the whole branch is cut [7, 2]. With OBDDs, it is possible to reduce an existing OBDD by traversing it depth-first and redirecting to the F node the paths representing inconsistent assignments [5]. However, this requires generating the non-reduced OBDD anyway.

Generating and handling conflict sets. Given an unsatisfiable assignment μ, we call a *conflict set* any unsatisfiable sub-assignment $\mu' \subset \mu$. (E.g., in Example 1 (2) and (3) are conflict sets for the assignment μ.) A key efficiency issue for MATH-SAT is the capability of MATHSOLVER to return the conflict set which has caused the inconsistency of an input assignment, and the capability of ASSIGN_ENUMERATOR to use this information to prune search.

For instance, both Belman-Ford algorithm and Simplex LP procedures can produce conflict sets [2, 15]. Semantic tableaux and DPLL can be enhanced by *mathematical backjumping* [9, 15, 2]: when MATHSOLVER(μ) returns a conflict set η, ASSIGN_ENUMERATOR 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 tree below. DPLL can be enhanced also with *learning* [15, 2]: the negation of the conflict set $\neg\eta$ is added in conjunction to the input formula, so that DPLL will never again generate an assignment containing the conflict set η.

Generating and handling derived assignments. Another efficiency issue for MATH-SAT is the capability of MATHSOLVER to produce an extra assignment η derived deterministically from a satisfiable input assignment μ, and the capability of ASSIGN_ENUMERATOR to use this information to narrow the search.

For instance, in the procedure presented in [2, 3], MATHSOLVER computes equivalence classes of real variables and performs substitutions which can produce further assignments. E.g., if $(v_1 = v_2), (v_2 = v_3) \in \mu$, $(v_1 - v_3 > 2) \notin \mu$ and μ is satisfiable, then MATHSOLVER(μ) finds that v_1 and v_3 belong to the same equivalence class and returns an extra assignment η containing $\neg(v_1 - v_3 > 2)$, which is unit-propagated away by DPLL.

Incrementality of MATHSOLVER. Another efficiency issue of MATHSOLVER is that of being *incremental*, so that to avoid restarting computation from scratch whenever it is given in input an assignment μ' s.t. $\mu' \supset \mu$ and μ has already proved satisfiable. (This happens, e.g., at the intermediate assignments checking steps.) Thus, MATHSOLVER should "remember" the status of the computation from one call to the other, whilst ASSIGN_ENUMERATOR should be able to keep track of the computation status of MATHSOLVER.

For instance, it is possible to modify a Simplex LP procedure so that to make it incremental, and to make DPLL call it incrementally after every unit propagation [15].

6 Implemented systems

6.1 Examples

Our framework captures a significant amount of existing procedure used in various application domains. We briefly recall some of them.

Omega [11] is a procedure used for dependence analysis of software. It is an integer programming algorithm based on an extension of Fourier-Motzkin variable elimination method. It handles boolean combinations of linear constraints by simply pre-computing the DNF of the input formula.

TSAT [1] is an optimized procedure for temporal reasoning able to handle sets of disjunctive temporal constraints. It integrates DPLL with a simplex LP tool, adding some form of forward checking and static learning.

LPSAT [15] is an optimized procedure for Math-formulas over linear real constraints, used to solve problems in the domain of resource planning. It accept only formulas with positive mathematical constraints. LPSAT integrates DPLL with an incremental simplex LP tool, and performs backjumping and learning.

SMV+QUAD-CLP [5] integrates OBDDs with a quadratic constraint solver to verify transition systems with integer data values. It performs a form of intermediate assignment checking.

DDD [10] are OBDD-like data structures handling boolean combinations of temporal constraints in the form $(x - z \leq 3)$, which are used to verify timed systems. They combine OBDDs with an incremental version of Belman-Ford minimal path and cycle detection algorithm.

6.2 A DPLL-based implementation of MATH-SAT

In [2, 3] we presented MATH-SAT, a decision procedure for Math-formulas over boolean and linear mathematical propositions over the reals. MATH-SAT uses as ASSIGN_ENUMERATOR an implementation of DPLL, and as MATHSOLVER a hierarchical set of mathematical procedures for linear constraints on real variables able to handle theories of increasing expressive power. The latter include a procedure for computing and exploiting equivalence classes from equality constraints like $(x = y)$, a Bellman-Ford minimal path algorithm with cycle detection for handling differences like $(x - y \leq 4)$, and a Simplex LP procedure for handling the remaining linear constraints. MATH-SAT implements and uses most of the tricks and optimizations described in Section 5.

In [2, 3] preliminary experimental evaluations were carried out on tests arising from temporal reasoning [1] and formal verification of timed systems [3]. In the first class of problems, we have compared our results with the results of the specialized procedure TSAT; although MATH-SAT is able to tackle a wider class of problems, it runs faster that the

TSAT solver, which is specialized to the problem class. In the second class, we have encoded bounded model checking problems for timed systems into the satisfiability of Math-formulas, and run MATH-SAT on them. It turned out that our approach was comparable in efficiency with well-established model checkers for timed systems, and significantly more expressive [3].

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. G. Audemard, P. Bertoli, A. Cimatti, A. Korniłowicz, and R. Sebastiani. A SAT Based Approach for Solving Formulas over Boolean and Linear Mathematical Propositions. 2002. Submitted to CADE'2002. Available at `http://www.science.unitn.it/~rseba/publist.html`.
3. G. Audemard, A. Cimatti, A. Korniłowicz, and R. Sebastiani. SAT-Based Bounded Model Checking for Timed Systems. 2002. Submitted to CAV'2002. Available at `http://www.science.unitn.it/~rseba/publist.html`.
4. R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
5. W. Chan, R. J. Anderson, P. Beame, 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.
6. M. Davis, G. Longemann, and D. Loveland. A machine program for theorem proving. *Journal of the ACM*, 5(7), 1962.
7. 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.
8. 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.
9. I. Horrocks and P. F. Patel-Schneider. FaCT and DLP. In *Procs. Tableaux'98*, number 1397 in LNAI, pages 27–30. Springer-Verlag, 1998.
10. J. Moeller, J. Lichtenberg, H. Andersen, and H. Hulgaard. Fully Symbolic Model Checking of Timed Systems using Difference Decision Diagrams. In *Electronic Notes in Theoretical Computer Science*, volume 23. Elsevier Science, 2001.
11. W. Pugh. The Omega Test: a fast and practical integer programming algoprithm for dependence analysis. *Communication of the ACM*, August 1992.
12. A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning*. Elsevier Science Publishers, 2001.
13. R. Sebastiani. Integrating SAT Solvers with Math Reasoners: Foundations and Basic Algorithms. Technical Report 0111-22, ITC-IRST, November 2001. Available at `http://www.science.unitn.it/~rseba/publist.html`.
14. R. M. Smullyan. *First-Order Logic*. Springer-Verlag, NY, 1968.
15. S. Wolfman and D. Weld. The LPSAT Engine & its Application to Resource Planning. In *Proc. IJCAI*, 1999.