UNIVERSITY
OF TRENTO

DEPARTMENT OF INFORMATION AND COMMUNICATION TECHNOLOGY

38050 Povo – Trento (Italy), Via Sommarive 14
http://www.dit.unitn.it

EFFICIENTLY INTEGRATING BOOLEAN REASONING AND MATHEMATICAL SOLVING

Gilles Audemard, Piergiorgio Bertoli, Alessandro Cimatti
Artur Kornilowicz and Roberto Sebastiani

January 2003

Technical Report # DIT-03-001

Article Submitted to Journal of Symbolic Computation

Efficiently Integrating Boolean Reasoning and Mathematical Solving * †

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,rseba}@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*

Abstract

Many real-world problems require the ability of reasoning efficiently on formulae which are boolean combinations of boolean and unquantified mathematical propositions. This task requires a fruitful combination of efficient boolean reasoning and mathematical solving capabilities. SAT tools and mathematical reasoners are respectively very effective on one of these activities each, but not on both. In this paper we present a formal framework, a generalized algorithm and architecture for integrating boolean reasoners and mathematical solvers so that they can efficiently solve boolean combinations of boolean and unquantified mathematical propositions. We describe many techniques to optimize this integration, and highlight the main requirements for SAT tools and mathematical solvers to maximize the benefits of their integration.

1. Motivation and goals

Many real-world problems require the ability of reasoning efficiently on formulae which are boolean combinations of boolean and unquantified mathematical propositions, on integer or real variables. (Noteworthy examples come from the domains of compilers design [Pugh, 1992], temporal reasoning [Armando et al., 1999], resource planning [Wolfman and Weld, 1999], automated verification of

*This paper extends the work presented in [Audemard et al., 2002b].

†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.

systems with numerical data [Chan et al., 1997] or of timed and hybrid systems [Moeller et al., 2001, Audemard et al., 2002c], software and protocol design and verification [Filliâtre et al., 2001, Stump et al., 2002].) This ability requires an efficient combination of *boolean reasoning* and *mathematical solving* capabilities.

From the viewpoint of boolean reasoning (SAT), in the last years we have witnessed an impressive advance in the efficiency of SAT techniques, which has allowed to solve previously intractable problems. Unfortunately, simple boolean expressions are not expressive enough for representing most of the real-world domains listed above. (Notice that encoding mathematical entities and operators into pure boolean expressions —e.g., by means of bitwise encodings for bounded integers or of boolean labeling techniques for simple real expressions— is typically very inefficient.)

From the viewpoint of mathematical solving, in the last years also mathematical solvers like computer-algebra systems and constraint solvers have very much improved both in expressivity and in efficiency, being able to solve classes of problems which were previously unsolvable or intractable. Unfortunately, mathematical solvers cannot handle efficiently problems involving heavy boolean search —or do not handle them at all— so that most of the real-world domains above are out of their reach too.

In this paper we present a formal framework, a generalized algorithm and architecture for integrating boolean reasoners and mathematical solvers so that they can efficiently solve boolean combinations of boolean and mathematical propositions. We describe many techniques to optimize this integration, and highlight the main requirements SAT tools and mathematical solvers must fulfill in order to achieve the maximum benefits from their integration. The work is inspired to the approach presented in Giunchiglia and Sebastiani [1996, 2000] for building domain-specific decision procedures on top of SAT tools, which has proved very effective in various problem domains like, e.g., modal and description logics [Giunchiglia and Sebastiani, 1996, 2000], temporal reasoning [Armando et al., 1999], resource planning [Wolfman and Weld, 1999].

The ultimate goal is to develop tools able to handle real-world problems in complex domains like those described above. From the viewpoint of boolean reasoning, SAT tools can be extended in such a way they can handle also mathematical concepts and operators. From the viewpoint of mathematical solving, computer algebra systems and constraint solvers can be enriched by very efficient boolean reasoning capabilities.

The paper is structured as follows. In Section 2 we describe formally the problem we are addressing. In Section 3 we present the formal framework on which the procedures are based. In Section 4 we present a generalized search procedure which combines boolean and mathematical solvers. In Section 5 we introduce some efficiency issues and we highlight the main requirements that boolean and mathematical solvers must fulfill in order to achieve the maximum benefits from their integration. In Section 6 we present our own implemented procedure. In Section 7 we briefly describe some other implemented systems which are captured by our framework. In Section 8 we present some related

work. For lack of space, we omit the proofs of all the theoretical results presented, which can be found in [Sebastiani, 2001].

2. The problem

We address the problem of checking the satisfiability of boolean combinations of primitive unquantified 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 in $\mathcal{D}$ and variables over the $\mathcal{D}$.

We call *math-terms* the unquantified mathematical expressions built up from 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-formulae* the mathematical formulae 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-formulae, 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-formulae. [‡]

Notationally, we use the lower case letters $t, t_1, ...$ to denote math-terms, and the Greek letters $\alpha, \beta, \varphi, \psi$ to denote math-formulae. 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-formulae respectively and preserves constants and arithmetical operators: $\mathcal{I}(A_i) \in \{\top, \bot\}$, 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,

[‡]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 some constraints like, e.g., $(v_1 \neq 0.0)$, $(v_1 \leq 5.0)$, etc.

[§]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 the arithmetic operator in $(\mathcal{I}_{\mathcal{D}}(t_1) + \mathcal{I}_{\mathcal{D}}(t_2))$. The same discourse holds for the constants $c_i \in \mathcal{C}$ and also for the operators $\{=, \neq, >, <, \geq, \leq\}$.

written "$\mathcal{I} \models \varphi$" ("$\mathcal{I}$ *satisfies* φ" or "$\mathcal{I}$ satisfies φ") is defined as follows:

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

We say that a math-formula φ is *satisfiable* if and only if there exists an interpretation $\mathcal{I}$ such that $\mathcal{I} \models \varphi$. E.g., if $\mathcal{D} = \mathbb{R}$, then $A_1 \rightarrow ((v_1 + 2v_2) \leq 4.5)$ is satisfied by an interpretation $\mathcal{I}$ such that $\mathcal{I}(A_1) = \top$, $\mathcal{I}(v_1) = 1.1$, 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 $\mathcal{I}$ such that $\mathcal{I} \models \varphi_1$. 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 $\models \varphi$ if and only if $\neg\varphi$ is unsatisfiable.

3. The formal framework

3.1. Basic definitions and results

Definition: 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, $\neg A_2$, $(v_1 + 3.0 \leq 4.0v_3)$, $\neg((3v_1/v_2) > 6)$. 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: We call a **total truth assignment** μ for a math-formula φ a set

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

such that every atom in $Atoms(\varphi)$ occurs as either a positive or a 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 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: 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-formulae φ_1, φ_2 of φ:

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

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.

(From now on, if not specified, when dealing with propositional satisfiability we do not distinguish between total and partial assignments.) We say that φ is *propositionally satisfiable* if and only if there exist an assignment μ such that $\mu \models_p \varphi$. Intuitively, if we consider a math-formula φ as a propositional formula 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 such that $\mu \models_p \varphi_1$. 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 $\models_p \varphi$ if and only if $\neg\varphi$ 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 3.1: Consider the following math-formula φ:

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

The truth assignment given by the underlined literals above is:

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

Notice that the two occurrences of $(3v_1 - 2v_2 \leq 3)$ in rows 3 and 5 of φ are both assigned $\top$. 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

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

do not have any satisfying interpretation. $\diamond$

Definition: 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

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

where each assignment μ_j is written as a conjunction of its elements.

$\mathcal{M}$ is complete in the sense that, for every total assignment η such that $\eta \models_p \varphi$, there exists $\mu_j \in \mathcal{M}$ such that $\mu_j \subseteq \eta$. 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 propositionally satisfying φ.

Definition: We say that a complete collection $\mathcal{M} = \{\mu_1, \ldots, \mu_n\}$ of assignments propositionally satisfying φ is **non-redundant** if for every $\mu_j \in \mathcal{M}$, $\mathcal{M} \setminus \{\mu_j\}$ is no more complete, it is **redundant** otherwise. $\mathcal{M}$ is **strongly non-redundant** if, for every $\mu_i, \mu_j \in \mathcal{M}$, $(\mu_i \wedge \mu_j)$ 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 3.2: 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 3.1: *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 $\mu_j \in \mathcal{M}$.*

3.2. Decidability and complexity

Having a math-formula φ, it is always possible to find a complete collection of satisfying assignments for φ (see later). Thus from Theorem 3.1 we have trivially the following fact.

PROPOSITION 3.1: *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., [Robinson and Voronkov, 2001]). Consequently, the satisfiability of math-formulae 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-formulae over non-linear constraints over $\mathbb{Z}$ is undecidable.

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

PROPOSITION 3.2: *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.

4. A generalized search procedure

Theorem 3.1 allows us to split the notion of satisfiability of a math-formula φ into two orthogonal components:

- a *purely boolean* component, consisting of the existence of a propositional model for φ;
- a *purely mathematical* component, consisting of the existence of an interpretation for a set of atomic mathematical propositions.

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

Definition: 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.)

Definition: 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: We call a **mathematical solver** a total function MATH-SOLVER which takes as input a set of atomic math-formulae μ and returns an interpretation satisfying μ, or *Null* if there is none.

```
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 :=MATH-SOLVER(μ);
    while ((I = Null) and (μ ≠ Null))
    if (I ≠ Null)
    then return True;                        /* a satisfiable assignment found */
    else return False;                       /* no satisfiable assignment found */
```

Figure 1: Schema of the general algorithm for MATH-SAT.

4.1. A generalized algorithm

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 $\mathcal{M} := \{\mu_1, .., \mu_n\}$ generated by ASSIGN-ENUMERATOR(φ), MATH-SAT invokes MATH-SOLVER 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 MATH-SOLVER terminates, then MATH-SAT terminates. Moreover, it follows from Theorem 3.1 that MATH-SAT is correct and complete if MATH-SOLVER 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 an arbitrary complete collection $\mathcal{M}$ of partial assignments propositionally satisfying φ, which is typically much smaller.

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, as far as we have a mathematical solver for it. This means that the expressivity of MATH-SAT, that is, the kind of math-formulae MATH-SAT can handle, depends only on the kind of sets of mathematical atomic propositions MATH-SOLVER can handle.

4.2. A generalized architecture

The general architectural schema for MATH-SAT is described in Figure 2. Notationally, φ is the input math-formula; Ψ is a boolean formula obtained by preprocessing φ —possibly CNF-izing it if ASSIGN-ENUMERATOR accepts only CNF formulae— and substituting in φ each atomic math-formula c_i with a new boolean variable A_i; $\mathcal{I}$ is an interpretation. μ is a truth value assignment to

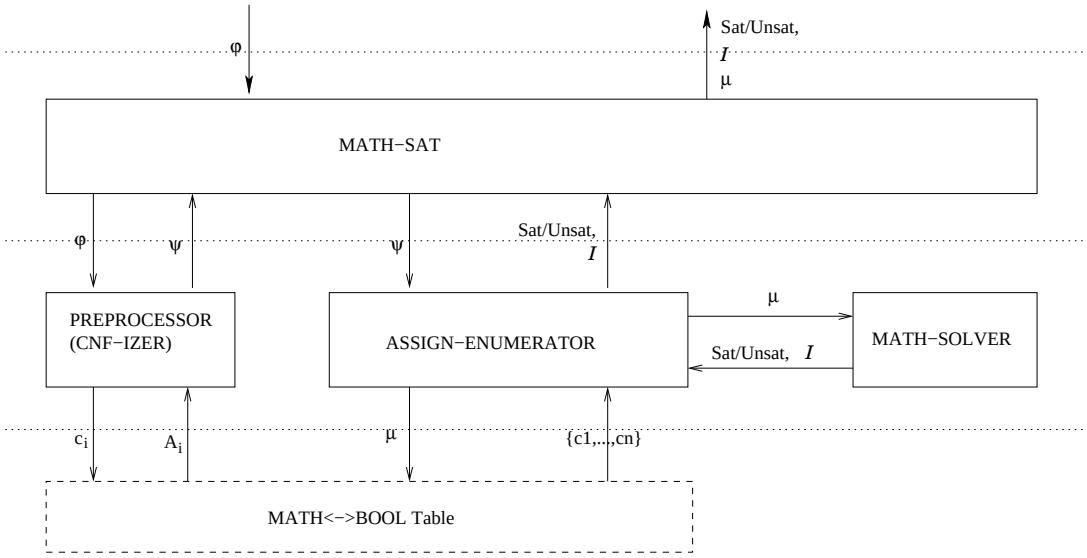


Figure 2: Schema of the general architecture for MATH-SAT.

the boolean variables of Ψ, and $\{c_1, ..., c_n\}$ is the corresponding set of atomic math-formulae. ¶

The PREPROCESSOR (CNF-IZER) receives in input a math-formula φ and returns a (CNF) boolean formula Ψ obtained by preprocessing φ (CNF-izing it) (see Giunchiglia and Sebastiani [1999]) and substituting in φ each atomic math-formula c_i with a new boolean variable A_i.

As a side effects, it produces a "MATH↔BOOL" table, which keeps a bijective relation between the atomic math-formulae of φ and their labeling boolean variables. Such table is used by ASSIGN-ENUMERATOR to convert each assignment μ into the corresponding set of atomic mathematical propositions, which is fed to MATH-SOLVER.

4.3. Suitable ASSIGN-ENUMERATORs

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

4.3.1. 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 formulae generate the same DNF modulo reordering.

¶Notice that $\{c_1, ..., c_n\}$ are *non-negated* atomic math-formulae: negations are inserted within the atoms like, e.g., $\neg(v_1 \leq c_1) \Longrightarrow (v_1 > c_1)$.

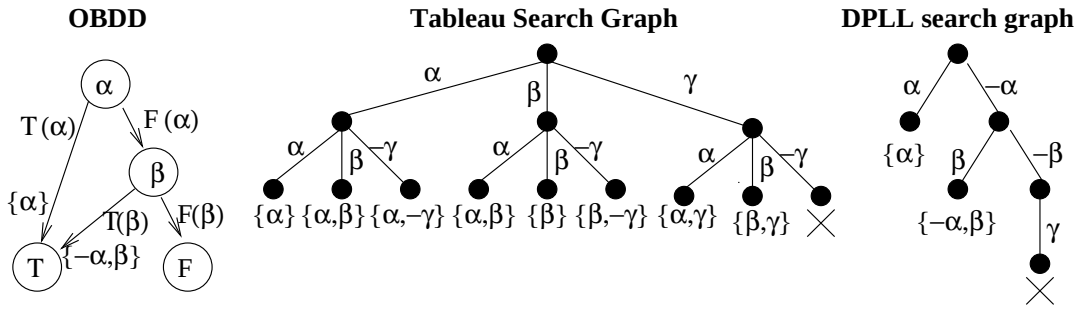


Figure 3: OBDD, Tableau and DPLL search graph for the formula $(\alpha \vee \beta \vee \gamma) \wedge (\alpha \vee \beta \vee \neg\gamma)$.

By Definition 3.4, 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 3.2, the set of assignments at point 2. and 3. are respectively the results of step (i) and (ii) above.

4.3.2. OBDD

A more effective normal form for representing a boolean formula if given by the *Ordered Binary Decision Diagrams (OBDDs)* [Bryant, 1986], 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 $v_i \rightarrow v_j$ 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 3 (left) the OBDD of the formula in Example 3.2 is represented. The paths to T are those given by the set of assignments at point 4. of Example 3.2.

4.3.3. Semantic tableaux.

A standard boolean solving technique is that of semantic tableaux [Smullyan, 1968]. 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:

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

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

```
boolean MATH-DPLL(formula φ, assignment & μ, interpretation & I)
    if (φ == ⊤) {                                          /* base        */
        I = MATH-SOLVER(μ) ;
        return (I ≠ Null) ; }
    if (φ == ⊥)                                            /* backtrack */
        return False;
    if {a literal l occurs in φ as a unit clause}          /* unit propagation  */
        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) );
```

Figure 4: Schema of an implementation of MATH-SAT based on DPLL.

- (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 such that $\mu \models_p \varphi$;
- ($\wedge$-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 3 (center) the search tree of a semantic tableau applied on the formula in Example 3.2 is represented. The solutions branches give rise to the redundant collection of assignments at point 2. of Example 3.2.

4.3.4. DPLL

Davis Putnam Longemann Loveland procedure (DPLL) [Davis et al., 1962] is probably the most commonly used boolean solving procedure. Figure 4 shows how it can be adapted to work as a boolean enumerator [Giunchiglia and Sebastiani, 1996, Sebastiani, 2001]. Given φ in input, MATH-DPLL tries to build recursively mathematically consistent assignments μ's propositionally satisfying φ. This is done adding at each step a new literal l to μ and simplifying φ, according to the following steps:

- (base) If $\varphi = \top$, then μ propositionally satisfies φ. Thus, if μ is satisfiable, then φ is satisfiable. Therefore MATH-DPLL invokes MATH-SOLVER(μ), which returns an interpretation for μ if it is satisfiable, *Null* otherwise. MATH-DPLL returns *True* in the first case, *False* otherwise.
- (backtrack) If $\varphi = \bot$, 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 $\top$.

Thus, MATH-DPLL is recursively invoked upon *assign(l, φ)* and the assignment obtained by adding l to μ. *assign(l, φ)* substitutes every occurrence of l in φ with $\top$ 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 upon *assign(l, φ)* and $\mu \cup \{l\}$. If the result is $False$, then MATH-DPLL is invoked upon *assign($\neg l, \varphi$)* and $\mu \cup \{\neg l\}$.

The key difference between MATH-DPLL and DPLL is in the "base" step: as DPLL needs finding only one satisfying assignment μ, it simply returns $True$.

The resulting set of assignments $DPLL(\varphi)$ is complete and strongly non-redundant [Sebastiani, 2001]. For instance, in Figure 3 (right) the search tree of DPLL applied on the formula in Example 3.2 is represented. The non closed branches give rise to the set of assignments at point 4. of Example 3.2.

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, whilst DPLL can choose each time the best variable to split. Even more important, DPLL can apply unit propagation if this is the case, OBDD cannot.

4.4. 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, some implementations 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 *assign(ψ, φ)* and $\mu \cup \{\psi\}$ [resp. *assign($\neg\psi, \varphi$)* 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 4.1: If we used DPLL+PL as ASSIGN-ENUMERATOR in MATH-SAT and gave in input the formula in Example 3.2, 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. Efficiency issues

In the schema in Figure 1, the efficiency of MATH-SAT does not depend only on the respective efficiency of its component procedures ASSIGN-ENUMERATOR and MATH-SOLVER. Other issues affect dramatically the global efficiency, like the *number of assignments* in the complete set generated by ASSIGN-ENUMERATOR, and the way ASSIGN-ENUMERATOR and MATH-SOLVER *interact*.

5.1. Efficiency of ASSIGN-ENUMERATOR

When we use a SAT solver like DPLL or OBDD as ASSIGN-ENUMERATOR, its efficiency as a SAT solver is not enough for guaranteeing the overall efficiency of MATH-SAT. We need some extra features, which we describe here.

5.1.1. Atoms' normalization

One potential source of inefficiency for the procedure for MATH-SOLVER 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. This causes the undesired generation of a potentially very big amount of intrinsically unsatisfiable assignments (like, e.g., $\{(v_1 < v_2), (v_1 \geq v_2), ...\}$).

To avoid these problems, it is wise to preprocess atoms so that to map semantically equivalent but syntactically different 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., $(v_1+(v_2+v_3))$, $((v_1+v_2)+v_3)) \Longrightarrow (v_1+v_2+v_3)$);
- *sort* (e.g., $(v_1 + v_2 \leq v_3 + 1)$, $(v_2 + v_1 - 1 \leq v_3) \Longrightarrow (v_1 + v_2 - v_3 \leq 1)$);
- *remove dual operators* (e.g., $(v_1 < v_2)$, $(v_1 \geq v_2) \Longrightarrow (v_1 < v_2)$, $\neg(v_1 < v_2)$).

5.1.2. Laziness of ASSIGN-ENUMERATOR.

We would rather MATH-SAT require polynomial space. As $\mathcal{M}$ can be exponentially big with respect to 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 in a "lazy" way, that is, one at a time.

When the input formula is consistent, laziness is crucial also for saving computation time. In fact, when a consistent assignment is found, a lazy ASSIGN-ENUMERATOR stops without generating the (up to exponentially many) other assignments in the complete set.

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

5.1.3. (Strong) 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 MATH-SOLVER 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 and DNF reduction are not good choices.

Non-redundant enumerators avoid generating partial assignments whose unsatisfiability is a propositional consequence of those already generated. 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.

5.2. Synergy between ASSIGN-ENUMERATOR and MATH-SOLVER

The way ASSIGN-ENUMERATOR and MATH-SOLVER interact is crucial for the overall efficiency of MATH-SAT, that is, an extremely efficient ASSIGN-ENUMERATOR integrated with an extremely efficient MATH-SOLVER can turn out to be dramatically inefficient unless the integration is done properly. For instance, ASSIGN-ENUMERATOR may do a huge amount of useless calls to MATH-SOLVER —e.g., on obviously inconsistent assignments— if it has no clue about the mathematical semantics of the truth assignments he generates; MATH-SOLVER can waste lots of time redoing the same computation at different calls if it keeps no information from one call to the other.

We describe here some techniques for maximizing the benefits of the interaction between ASSIGN-ENUMERATOR and MATH-SOLVER.

5.2.1. Intermediate assignment checking

If an assignment μ' is unsatisfiable, then all its extensions are unsatisfiable. Thus, when the unsatisfiability of 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 MATH-SOLVER 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 MATH-SOLVER is invoked on an intermediate assignment μ': if μ' is inconsistent, the whole branch is cut [Giunchiglia and Sebastiani, 1996, Audemard et al., 2002a]. 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 [Chan et al., 1997, Moeller et al., 2001]. However, this requires generating the non-reduced OBDD anyway.

5.2.2. Mathematical Backjumping and Learning.

Given an unsatisfiable assignment μ, we call a *conflict set* any unsatisfiable sub-assignment $\mu' \subset \mu$. (E.g., in Example 3.1 (2) and (3) are conflict sets for the assignment μ.) A key efficiency issue for MATH-SAT is the capability of MATH-SOLVER 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 [Audemard et al., 2002a, Wolfman and Weld, 1999]. Semantic tableaux and DPLL can be enhanced by a technique called *mathematical backjumping* [Horrocks and Patel-Schneider, 1998, Wolfman and Weld, 1999, Audemard et al., 2002a]: when MATH-SOLVER(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* [Wolfman and Weld, 1999, Audemard et al., 2002a]: 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.

5.2.3. Generating and handling derived assignments.

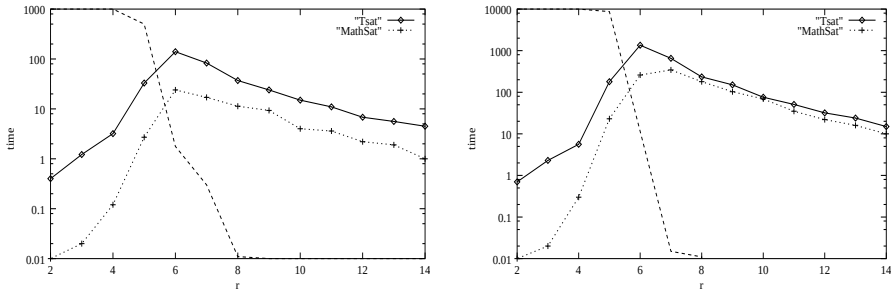
Another efficiency issue for MATH-SAT is the capability of MATH-SOLVER to produce an extra assignment η derived deterministically from a satisfiable input assignment μ, and the capability of ASSIGN-ENUMERATOR to use this information to narrow the search.

For instance, in the procedure presented in [Audemard et al., 2002a,c], MATH-SOLVER 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 MATH-SOLVER(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.

5.2.4. Incrementality of MATH-SOLVER.

Another efficiency issue of MATH-SOLVER is that of being *incremental*, so that to avoid restarting computation from scratch whenever it is given in input an assignment μ' such that $\mu' \supset \mu$ and μ has already proved satisfiable. (This happens, e.g., at the intermediate assignments checking steps.) Thus, MATH-SOLVER 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 MATH-SOLVER.

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 [Wolfman and Weld, 1999].



25 vars 30 vars

Figure 5: Comparison between TSAT and MATH-SAT [Audemard et al., 2002a]. $k = 2$, $n = 25, 30$, $L = 100$, $r := m/n$ in $[2, ..., 14]$. 100 sample formulae per point. Median CPU times (secs). Background: satisfiability rate.

	MATH-SAT		MATH-SAT,Sym		DDD		Uppal		Kronos		Red		Red,Sym	
N	Time	Size	Time	Size	Time	Size	Time	Size	Time	Size	Time	Size	Time	Size
2	0.02	2.7	0.02	2.7	0.09	106	0.01	1.5	0.01	0.6	0.05	1.9	0.04	1.9
3	0.05	2.9	0.04	2.9	0.11	106	0.01	1.7	0.01	0.8	0.23	2.0	0.19	2.0
4	0.09	3.0	0.08	3.0	0.14	106	0.02	1.9	0.02	2.2	1.00	2.1	0.70	2.1
5	0.20	3.2	0.16	3.2	0.24	106	0.21	1.9	0.09	19	3.70	2.2	2.00	2.4
6	0.60	3.7	0.23	3.7	0.47	106	3.44	6.7	0.39	236	12.00	2.7	5.20	3.1
7	3.20	4.2	0.36	4.2	1.30	106	153	54		-	38	4.0	12	4.7
8	29	4.9	0.52	4.9	3.96	106	-				121	7.6	26	7.8
9	343	5.9	0.75	5.9	14	106					416	16.6	49	13.3
10	3331	6.5	1.01	6.5	62	106					1382	39	90	23
11	-		1.39	7.0	691	106					-		157	38
12			1.89	7.5		-							266	63
13			2.44	8.2									439	100
14			3.24	8.9									709	155
15			4.11	9.7									1118	225
16			5.10	10.7									1717	342
17			6.30	11.7									2582	492
18			8.00	12.9									-	
19			9.50	14.2										

Table 1: Verification of a reachability property for Fischer's mutual-exclusion protocol (time in seconds, size in MB) [Audemard et al., 2002c]. N is the number of concurrent processes. MATH-SAT is compared against the model checkers DDD, Uppal, Kronos, Red. Two different encodings into math-formulae, basic and symmetry-exploiting (Sym), are used.

6. A DPLL-based implementation of MATH-SAT

In [Audemard et al., 2002a,c] we presented MATH-SAT, a decision procedure for math-formulae over boolean and linear mathematical propositions over the reals. In [Audemard et al., 2002c] we presented a new approach for solving bounded model checking problems for real-time systems by encoding them into math-formulae and hence feeding them to MATH-SAT. (Intuitively, boolean variables encode the discrete part of the system tested, whilst linear constraints on real variables, representing absolute time and clock values, encode the timed part.)

MATH-SAT uses as ASSIGN-ENUMERATOR an implementation of DPLL based on the SIM library [Giunchiglia et al., 2001], and as MATH-SOLVER a combination of symbolic and numeric mathematical procedures for linear mathematical propositions on real variables. The latter include a procedure for computing and exploiting equivalence classes from equality constraints like $(x = y)$, a Bellman-Ford algorithm with negative cycle detection for handling differences like $(x - y \leq 4)$, and a Simplex LP procedure for generic linear constraints.

The different procedures are organized in layers of increasing expressive power, each layer coming into play only when needed. MATH-SAT implements and uses the tricks and optimizations, plus others, described in Section 5. Technical details can be found in [Audemard et al., 2002a]. MATH-SAT is available at `http://www.dit.unitn.it/~rseba/Mathsat.html`.

In [Audemard et al., 2002a,c] experimental evaluations were carried out on tests arising from temporal reasoning [Armando et al., 1999] and formal verification of real-time systems. In the first class of problems, we compared our results with the results of the specialized procedure TSAT, on the random problems presented in [Armando et al., 1999]. (They are random math-formulae in the form $\bigwedge_{i=1}^{m}\bigvee_{j=1}^{2}(v_{1_{ij}} - v_{2_{ij}} \leq c_{ij})$, $v_{k_{ij}}$ and c_{ij} being respectively real variables picked uniformly among $\{v_1, ..., v_n\}$ and integer constants picked uniformly in $[-L, ..., L]$.) 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. The plots are reported in Figure 5. (See [Audemard et al., 2002a] for details.)

In the second class, we encoded bounded model checking problems for real-time systems into the satisfiability of math-formulae, run MATH-SAT on them, and compared the results with those of well-established model checkers for timed systems. An example is reported in Table 1. It turned out that our approach is comparable in time efficiency with the well-established model checkers for timed systems and, unlike all other approaches, requires only polynomial memory. (See [Audemard et al., 2002c] for details.)

7. Implemented systems

Our framework captures a significant amount of existing procedure used in various application domains. These procedures either are purely symbolic or combine symbolic and numeric techniques. We briefly recall some of them.

Omega [Pugh, 1992] is a symbolic+numeric procedure used for dependence analysis of software. It is an integer programming algorithm based on Fourier-Motzkin variable elimination method. It handles boolean combinations of linear constraints by pre-computing the DNF of the input formula.

PtautEq [Armando and Giunchiglia, 1993] is a purely symbolic procedure which handles boolean combinations of boolean variables and equalities between first-order variables, which was embedded in the GETFOL system. It combines a variant of DPLL with an ad-hoc solver for sets of equalities.

SMV+QUAD-CLP [Chan et al., 1997] is an incomplete symbolic+numeric procedure integrating OBDDs with a quadratic constraint solver to verify transition systems with integer data values. It performs a form of intermediate assignment checking.

TSAT [Armando et al., 1999] is an optimized symbolic+numeric 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 [Wolfman and Weld, 1999] is an optimized symbolic+numeric procedure for math-formulae over linear real constraints, used to solve problems in the domain of resource planning. It accept only formulae with positive mathematical constraints. LPSAT integrates DPLL with an incremental simplex LP tool, and performs backjumping and learning.

DDD's [Moeller et al., 2001] 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 algorithm.

ICS [Filliâtre et al., 2001] is a mostly symbolic decision procedure for combined theories, including theory of arrays, bitvectors, lists and inductive datatypes, linear arithmetic over the integers. Very recently (2002) it has been integrated with the DPLL solver CHAFF [Moskewicz et al., 2001].

CVC [Stump et al., 2002] is a symbolic+numeric decision procedure for combined theories, including theory of arrays, inductive datatypes, linear arithmetic over the reals. It combines, among others, the DPLL solver CHAFF with a Fourier-Motzkin procedure.

Notice that Omega, SMV+QUAD-CLP and DDD inherit from DNF and OBDDs the drawback of requiring exponential space in worst case.

8. Related work

Since the seminal work by Nelson and Oppen [1979] and Shostak [1979], many papers have been presented on the integration decision procedures for different theories, including linear arithmetic. However, in these approaches very little effort has been made to handle efficiently the boolean component of reasoning. Only very recently (2002) the integration of procedures like ICS and CVC with SAT solvers has been investigated.

From the strict viewpoint of SAT systems, a related but different approach which is not listed in Section 7 is that proposed by Strichman [2002]. He presents a reduction from linear math-formulae on the reals into pure boolean formulae, which are then fed to a SAT solver. The reduction works by substituting each inequality with a distinct boolean atom, and by adding new boolean variables and constraints which mimic the application of the Fourier-Motzkin elimination technique. Unfortunately, the resulting formulas blow up in size in worst-case.

From the viewpoint of computer algebra, a related but different approach is that of the REDLOG system [Dolzmann and Sturm, 1997, Sturm, 2002]. REDLOG is an extension of the computer algebra system REDUCE, featuring many symbolic reasoning algorithms on specific first-order mathematical theories. REDLOG is based on efficient quantifier elimination procedures, enhanced with tools for manipulating formulae (including algebraic simplification, normal form computation, linear optimization). Although REDLOG is not a satisfiability procedure in the strict sense, its degree-restricted real quantifier elimination procedure can solve the consistency problem for many math-formulae, including linear formulae on the reals, by eliminating the existential closure of the input formulae.

REDLOG's approach is very general (e.g., Weispfenning [1999] showed that the mixed linear theory of the reals and the integers can be decided by quantifier elimination; moreover, REDLOG's simplification techniques and heuristics allow for eliminating many non-linear formulae). Nevertheless, even for simple linear math-formulae on the reals, there is no guarantee that REDLOG will find a solution within polynomial memory [Dolzmann, 2002].

References

A. Armando, C. Castellini, and E. Giunchiglia. SAT-based procedures for temporal reasoning. In *Proc. European Conference on Planning, CP-99*, 1999.

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.

G. Audemard, P. Bertoli, A. Cimatti, A. Korniłowicz, and R. Sebastiani. A SAT Based Approach for Solving Formulas over Boolean and Linear Mathematical Propositions. In *Proc. CADE'2002.*, LNAI. Springer, July 2002a.

G. Audemard, P. Bertoli, A. Cimatti, A. Korniłowicz, and R. Sebastiani. Integrating Boolean and Mathematical Solving: Foundations, Basic Algorithms and Requirements. In *Proc. AISC+CALCULEMUS'2002.*, LNAI. Springer, 2002b.

G. Audemard, A. Cimatti, A. Korniłowicz, and R. Sebastiani. SAT-Based Bounded Model Checking for Timed Systems. In *Proc. FORTE'02.*, LNCS. Springer, November 2002c.

R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

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.

M. Davis, G. Longemann, and D. Loveland. A machine program for theorem proving. *Journal of the ACM*, 5(7), 1962.

A. Dolzmann, 2002. Personal communication.

A. Dolzmann and T. Sturm. REDLOG: Computer algebra meets computer logic. *ACM SIGSAM Bulletin*, 31(2), June 1997.

J. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: integrated canonizer and solver. In *Proc. CAV'2001*, July 2001.

E. Giunchiglia, M. Narizzano, A. Tacchella, and M. Vardi. Towards an Efficient Library for SAT: a Manifesto. In *Proc. SAT 2001.* Elsevier Science., 2001.

E. Giunchiglia and R. Sebastiani. Applying the Davis-Putnam procedure to non-clausal formulas. In *Proc. AI*IA'99*, LNAI. Springer, 1999.

F. Giunchiglia and R. Sebastiani. A SAT-based decision procedure for ALC. In *Proc. KR'96*, Cambridge, MA, USA, November 1996.

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.

I. Horrocks and P. F. Patel-Schneider. FaCT and DLP. In *Procs. Tableaux'98*, LNAI, pages 27–30. Springer, 1998.

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.

M. W. Moskewicz, C. F. Madigan, Y. Z., L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference*, 2001.

C. G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *TOPLAS*, 1(2):245–257, 1979.

W. Pugh. The Omega Test: a fast and practical integer programming algorithm for dependence analysis. *Communication of the ACM*, August 1992.

A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning*. Elsevier Science Publishers, 2001.

R. Sebastiani. Integrating SAT Solvers with Math Reasoners: Foundations and Basic Algorithms. Technical Report 0111-22, ITC-IRST, November 2001. Available at `http://www.dit.unitn.it/~rseba/publist.html`.

R. Shostak. A Pratical Decision Procedure for Arithmetic with Function Symbols. *Journal of the ACM*, 26(2):351–360, 1979.

R. M. Smullyan. *First-Order Logic*. Springer-Verlag, NY, 1968.

O. Strichman. On Solving Presburger and Linear Arithmetic with SAT. In *Proc. FMCAD 2002*, page 11, November 2002.

A. Stump, C. W. Barret, and D. L. Dill. CVC: A Cooperative Validity Checker. In *Proc. CAV'02*, LNCS. Springer, July 2002.

T. Sturm. Integration of quantifier elimination with constraint logic programming. In *Proc AISC+CALCULEMUS'2002*, LNAI. Springer, june 2002.

V. Weispfenning. Mixed real-integer linear quantifier elimination. In *Proc. ISSAC'99*. ACM Press, 1999.

S. Wolfman and D. Weld. The LPSAT Engine & its Application to Resource Planning. In *Proc. IJCAI*, 1999.