Journal of Automated Reasoning (2005) 35: 265–293
DOI: 10.1007/s10817-005-9004-z

MATHSAT: Tight Integration of SAT and Mathematical Decision Procedures*

MARCO BOZZANO[1], ROBERTO BRUTTOMESSO[1], ALESSANDRO CIMATTI[1], TOMMI JUNTTILA[2], PETER VAN ROSSUM[1], STEPHAN SCHULZ[3], and ROBERTO SEBASTIANI[4]
[1]*ITC-IRST, via Sommarive 18, 38050, Povo, Trento, Italy.*
e-mail: {bozzano, bruttomesso, cimatti, vanrossum}@itc.it
[2]*Helsinki University of Technology, P.O. Box 5400, FIN-02015 TKK, Helsinki, Finland.*
e-mail: tommi.junttila@tkk.fi
[3]*Università di Verona, Strada le Grazie 15, 37134, Verona, Italy. e-mail: schulz@eprover.org*
[4]*DIT, Università di Trento, via Sommarive 14, 38050, Povo, Trento, Italy.*
e-mail: roberto.sebastiani@dit.unitn.it

Abstract. Recent improvements in propositional satisfiability techniques (SAT) made it possible to tackle successfully some hard real-world problems (e.g., model-checking, circuit testing, propositional planning) by encoding into SAT. However, a purely Boolean representation is not expressive enough for many other real-world applications, including the verification of timed and hybrid systems, of proof obligations in software, and of circuit design at RTL level. These problems can be naturally modeled as satisfiability in linear arithmetic logic (LAL), that is, the Boolean combination of propositional variables and linear constraints over numerical variables. In this paper we present MATHSAT, a new, SAT-based decision procedure for LAL, based on the (known approach) of integrating a state-of-the-art SAT solver with a dedicated mathematical solver for LAL. We improve MATHSAT in two different directions. First, the top-level line procedure is enhanced and now features a tighter integration between the Boolean search and the mathematical solver. In particular, we allow for theory-driven backjumping and learning, and theory-driven deduction; we use static learning in order to reduce the number of Boolean models that are mathematically inconsistent; we exploit problem clustering in order to partition mathematical reasoning; and we define a stack-based interface that allows us to implement mathematical reasoning in an incremental and backtrackable way. Second, the mathematical solver is based on layering; that is, the consistency of (partial) assignments is checked in theories of increasing strength (equality and uninterpreted functions, linear arithmetic over the reals, linear arithmetic over the integers). For each of these layers, a dedicated (sub)solver is used. Cheaper solvers are called first, and detection of inconsistency makes call of the subsequent solvers superfluous. We provide a through experimental evaluation of our approach, by taking into account a large set of previously proposed benchmarks. We first investigate the relative benefits and drawbacks of each proposed technique by

* This work has been partly supported by ISAAC, a European-sponsored project, contract no. AST3-CT-2003-501848; by ORCHID, a project sponsored by Provincia Autonoma di Trento; and by a grant from Intel Corporation. The work of T. Junttila has also been supported by the Academy of Finland, project 53695. S. Schulz has also been supported by a grant of the Italian Ministero dell'Istruzione, dell'Università e della Ricerca and the University of Verona.

comparison with respect to a reference option setting. We then demonstrate the global effectiveness of our approach by a comparison with several state-of-the-art decision procedures. We show that the behavior of MATHSAT is often superior to its competitors, both on LAL and in the subclass of difference logic.

Key words: satisfiability module theory, integrated decision procedures, linear arithmetic logic, propositional satisfiability.

1. Motivations and Goals

Many practical domains of reasoning require a degree of expressiveness beyond propositonal logic. For instance, timed and hybrid systems have a discrete component as well as a dynamic evolution of real variables; proof obligations arising in software verification are often Boolean combinations of constraints over integer variables; circuits described at the register transfer level, even though expressible via Booleanization, might be easier to analyze at a higher level of abstraction (see e.g., [12]). The verification problems arising in such domains can often be modeled as satisfiability in Linear Arithmetic Logic (LAL), that is, the Boolean combination of propositional variables and linear constraints over numerical variables. Because of its practical relevance, LAL has attracted a lot of interest, and several decision procedures (e.g., SVC [15], ICS [18, 23], CVCLITE [7, 15], UCLID [35, 43], HDPLL [31]) are able to deal with it.

In this paper, we propose a new decision procedure for the satisfiability of LAL, both for the real-valued and for the integer-valued case. We start from a well-known approach, previously applied in MATHSAT [3, 27] and in several other systems [2, 7, 15, 18, 20, 23, 42]: a propositional SAT procedure, modified to enumerate propositional assignments for the propositional abstraction of the problem, is integrated with dedicated theory deciders, used to check consistency of propositional assignments with respects to the theory. We extend this approach by improving (1) the top-level procedure, and (2) the mathematical reasoner.

The *top-level procedure* features a tighter integration between the Boolean search and the mathematical solver. First, we allow for theory conflict-driven backjumping (i.e., sets of inconsistent constraints identified in the mathematical solver are used to drive backjumping and learning at the Boolean level) and theory deduction (i.e., when possible, assignments for unassigned theory atoms are automatically inferred from the current partial assignment). Both theory conflicts and theory deductions are learned as clauses codifying the relationships between mathematical atoms at the Boolean level; subsequent search will thus avoid the generation of Boolean assignments that are not mathematically consistent. Second, we suggest a systematic use of *static learning*, that is, the *a priori* encoding of some basic mathematical facts at the Boolean level before the Boolean search. This will stop many inconsistent assignments from ever being enumerated. A moderate increase in the size of the problem is often compensated by significant speedups in performance. In this way MATHSAT settles in the

middle ground between the "eager" approach, where mathematical facts are discovered during the search, and the "lazy approaches" approach (e.g., [39, 43]), where a very large number of facts may be required in order to lift mathematical reasoning to Boolean reasoning. Third, we define a *stack-based interface* between the Boolean level and the mathematical level, which enables the top level to add constraints, set points of backtracking, and backjump, in order to exploit the fact that increasingly larger sets of constraints are analyzed while extending a Boolean model. As a result, the mathematical reasoner can be incremental and backtrackable and can exploit previously derived information rather than restarting from scratch at each call. Finally, we consider that mathematical reasoning is, in many practical cases, performed on the disjoint union of several subtheories (or *clusters*). Therefore, rather than solving the problem with a single, monolithic mathematical solver, we use a separate instance of the mathematical solver for each independent cluster.

The main idea underlying the *mathematical solver* for linear arithmetic is that it is *layered*, that is, implemented as a hierarchy of solvers for theories of increasing strength. The consistency of (partial) assignments is checked first in the logic of equality and uninterpreted function (EUF), then in difference logics, then in linear arithmetic over the reals, and then in linear arithmetic over the integers (if needed by the problem). The rationale is that cheaper, more abstract solvers are called first. If unsatisfiability at a more abstract level is detected, this is sufficient to prune the search.

We provide a thorough experimental evaluation of our approach, based on a large set of benchmarks previously proposed in the literature. We first show the respective merits of each of the proposed optimizations, comparing different configurations of MATHSAT with respect to a "golden setting", and we show to which extent each of the improvements impacts performance. Then we compare MATHSAT against the state-of-the-art systems (ICS, CVCLITE, and UCLID) on general LAL problems. We show that our approach is able to deal efficiently with a wide class of problems, with performance comparable with and often superior to the other systems. We also compare MATHSAT against the specialized decision procedures DLSAT and TSAT++ on the subclass of difference logics.

This paper is structured as follows. In Section 2 we define linear arithmetic logic. In Section 3 we describe the basic MATHSAT approach, and in Section 4 we present the enhanced algorithm. In Section 5 we describe the ideas underlying the mathematical solver. In Section 6 we described the implementation of the MATHSAT system. In Section 7 we present the result of the experimental evaluation. In Section 8 we discuss some related work; and in Section 9 we draw some conclusions and outline the directions for future work.

This paper updates and extends the content and results presented in a much shorter conference paper [11].

2. Background: Linear Arithmetic Logic

Let $\mathbb{B} := \{\bot, \top\}$ be the domain of Boolean values. Let $\mathbb{R}$ and $\mathbb{Z}$ be the domains of real and integer numbers, respectively, and let $\mathcal{D}$ denote either of them. By *math-terms* over $\mathcal{D}$ we denote the linear mathematical expressions built on constants, variables, and arithmetical operators over $\mathcal{D}$. Examples of math-terms are constants $c_i \in \mathcal{D}$, variables v_i over $\mathcal{D}$, possibly with coefficients (i.e., $c_i v_j$), and applications of the arithmetic operators + and $-$ to math-terms. *Boolean atoms* are proposition A_i, from $\mathbb{B}$. *Mathematical atoms* are formed by the application of the arithmetic relations $=, \neq, >, <, \geq, \leq$ to math-terms. Unspecified *atoms* can be either Boolean or mathematical. By *math-formulas* we denote atoms and their combinations through the standard Boolean connectives $\wedge$, $\neg$, $\vee$, $\rightarrow$, $\leftrightarrow$. For instance, $A_1 \wedge((v_1 + 5) \leq 2v_3)$ is a math-formula on either $\mathbb{R}$ or $\mathbb{Z}$. A *literal* is either an atom (a *positive* literal) or its negation (a *negative* literal). Examples of literals are A_1, $\neg A_2$, $(v_1 + 5v_2 \leq 2v_3 - 2)$, $\neg(2v_1 - v_2 = 5)$. If l is a negative literal $\neg\psi$, then by "$\neg l$" we denote ψ rather than $\neg\neg\psi$. We denote the set of all atoms in ϕ by *Atoms*(ϕ), and the subset of mathematical atoms by *MathAtoms*(ϕ).

An *interpretation* in $\mathcal{D}$ is a mapping I which assigns values in $\mathcal{D}$ to variables and truth values in $\mathbb{B}$ to Boolean atoms. Given an interpretation, math-terms and math-formulas are given values $\mathcal{D}$ and in $\mathbb{B}$, respectively, by interpreting constants, arithmetical operators and Boolean connectives according to their standard (arithmetical or logical) semantics. We write $I(\phi)$ for the truth value of ϕ under the interpretation I, and similarly $I(t)$ for the domain value of the math-term t. We say that I *satisfies* a math-formula ϕ, written $I \models \phi$, iff $I(\phi) = \top$. For example, the math-formula $\varphi := (A_1 \rightarrow (v_1 - 2v_2 \geq 4)) \wedge (\neg A_1 \rightarrow (v_1 = v_2 + 3))$ is satisfied by an interpretation I in $\mathbb{Z}$ s.t. $I(A_1) = \top$, $I(v_1) = 8$, and $I(v_2) = 1$.

We say that a math-formula φ is satisfiable in $\mathcal{D}$ if there exists an interpretation in $\mathcal{D}$ which satisfies φ. The problem of checking the satisfiability of math-formulas is NP-hard, since standard Boolean formulas are a strict subcase

```
function MATHSAT (Math-formula φ, interpretation & I)
    return MATHDPLL (M2B(φ), {}, I);

function MATHDPLL (Boolean-formula φ, assignment & μ,
                   interpretation & I)
    if (φ == ⊤)                                  /* base       */
        then return MATHSOLVE (B2M(μ), I) ;
    if (φ == ⊥)                                  /* backtrack  */
        then return Unsat;
    if {l occurs in φ as a unit clause}          /* unit prop. */
        then return MATHDPLL (assign(l, φ), μ ∪ {l}, I);
    if (MATHSOLVE (B2M(μ), I) == Unsat)          /* early pruning */
        then return Unsat;
    l := choose-literal(φ);                      /* split      */
    if ( MATHDPLL (assign(l, φ), μ ∪ {l}, I) == Sat )
        then return Sat;
        else return MATHDPLL (assign(¬l, φ), μ ∪ {¬l}, I);
```

Figure 1. High level view of the MATHSAT algorithm.

of math-formulas (this means theoretically "at least as hard" as standard Boolean satisfiability, but in practice it turns out to be much harder).

A total (resp., partial) *truth assignment* for a math-formula ϕ is a function μ from all (resp., a subset of) the atoms of ϕ to truth values. We represent a truth assignment as a set of literals, with the intended meaning that positive and negative literals represent atoms assigned to true and to false, respectively. We use the notation $\mu = \{\alpha_1, \ldots, \alpha_N, \neg\beta_1, \ldots, \neg\beta_M, A_1, \ldots, A_R, \neg A_{R+1}, \ldots, \neg A_S\}$, where $\alpha_1, \ldots, \alpha_N, \beta_1, \ldots, \beta_M$ are mathematical atoms and $A_1, \ldots, A_S$ are Boolean atoms. We say that μ *propositionally satisfies* ϕ, written $\mu \models_p \phi$, iff it makes ϕ evaluate to true. We say that an interpretation I satisfies a truth assignment μ iff I satisfies all the elements of μ; if there exists an (resp., no) interpretation that satisfies an assignment μ, then μ is said LAL-satisfiable (resp., LAL-unsatisfiable). The truth assignment $\{A_1, (v_1 - 2v_2 \geq 4), \neg(v_1 = v_2 + 3)\}$ propositionally satisfies $(A_1 \rightarrow (v_1 - 2v_2 \geq 4)) \wedge (\neg A_1 \rightarrow (v_1 = v_2 + 3))$, and it is satisfied by I s.t. $I(A_1) = \top$, $I(v_1) = 8$, and $I(v_2) = 1$.

EXAMPLE 2.1 Consider the following math-formula φ:

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

The truth assignment μ corresponding to the underlined literals is

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

(Notice that μ is a partial assignment, because it assigns truth values only to a subset of the atoms of φ.) μ propositionally satisfies φ as it sets to true one literal of every disjunction in φ. Notice that μ is not LAL-satisfiable – in fact, neither of the following subassignments of μ has a satisfying interpretation:

$$\{\neg(2v_2 - v_3 > 2), (3v_1 - 2v_2 \leq 3), \neg(3v_1 - v_3 \leq 6)\} \tag{1}$$

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

Given a LAL-unsatisfiable assignment μ, we call a *conflict set* any LAL-unsatisfiable subassignment $\mu' \subseteq \mu$; we say that μ' is a *minimal* conflict set if all

subsets of μ' are LAL-consistent. For example, both (1) and (2) are minimal conflict sets of μ.

3. The MathSAT Algorithm: Basics

A much simplified, recursive representation of the basic MathSAT procedure is outlined in Figure 1. MathSAT takes as input a math-formula ϕ, and (by reference) any empty interpretation I. Without loss of generality, ϕ is assumed to be in conjunctive normal form (CNF). MathSAT returns $\top$ if ϕ is LAL-satisfiable (with I containing a satisfying interpretation), and $\bot$ otherwise. MathSAT invokes MathDpll passing as arguments the Boolean formula $\varphi := \mathcal{M}2\mathcal{B}(\phi)$ and (by reference) an empty assignment for φ and the empty interpretation I.

We introduce a bijective function $\mathcal{M}2\mathcal{B}$ (for "Math-to-Boolean"), also called *boolean abstraction* function, that maps Boolean atoms into themselves, math-atoms into fresh Boolean atoms – so that two atom instances in φ are mapped into the same Boolean atom iff they are syntactically identical and distributes over sets and Boolean connectives. Its inverse function $\mathcal{B}2\mathcal{M}(\mu)$ (for "Boolean-to-Math") is called *refinement*, respectively. Both functions can be implemented efficiently, so that they require a small constant time for mapping one atom.

MathDpll tries to build an assignment μ satisfying φ, such that its refinement is satisfiable in LAL, and the interpretation I satisfies $\mathcal{B}2\mathcal{M}(\mu)$ (and ϕ). This is done recursively, with a variant of DPLL modified to enumerate assignments, and trying to refine them according to LAL. In particular:

Base. If $\varphi == \top$, then μ propositionally satisfies $\mathcal{M}2\mathcal{B}(\phi)$. In order to check whether μ is LAL-satisfiable, which shows that φ is LAL-satisfiable, MathDpll invokes the linear mathematical solver MathSolve on the refinement $\mathcal{B}2\mathcal{M}(\mu)$, and returns a *Sat* or *Unsat* value accordingly.

Backtrack. If $\varphi == \bot$, then μ has led to a propositional contradiction. Therefore MathDpll returns *Unsat* and backtracks.

Unit. If a literal l occurs in φ as a unit clauses, then l must be assigned a true value. Thus MathDpll is invoked recursively with the formula returned by *assign* (l, φ) and the assignment obtained by adding l to μ as arguments. *assign* (l, φ) substitutes every occurrence of l in φ with $\top$ and propositionally simplifies the result.

Early pruning. MathSolve is invoked on (the refinement of) the current assignment μ. If this is found unsatisfiable, then there is no need to proceed, and the procedure backtracks.

Split. If none of the above situations occurs, then *choose-literal* (φ) returns an unassigned l according to some heuristic criterion. Then MATHDPLL is first invoked recursively with arguments *assign*(l, φ) and $\mu \cup \{l\}$. If the result is *Unsat*, then MATHDPLL is invoked with argument *assign* ($\neg l$, φ) and $\mu \cup \{\neg l\}$.

4. The MATHSAT Algorithm: Enhancements

The algorithm presented in the previous section is oversimplified for explanatory purposes. It can be easily adapted to deal with advanced SAT solving techniques such as splitting heuristics, two-literals watching, and restarts (see [44] for an overview). This section describes several enhancement that have been made to the interplay between the Boolean and mathematical solvers.

4.1. THEORY-DRIVEN BACKJUMPING AND LEARNING

When MATHSOLVE finds the assignment μ to be LAL-unsatisfiable, it also returns a conflict set η causing the unsatisfiability. This enables MATHDPLL to backjump in its search to the most recent branching point in which at least one literal $l \in \eta$ is not assigned a truth value, pruning the search space below. We call this technique *theory-driven backjumping*. Clearly, its effectiveness strongly depends on the quality of the conflict sets generated.

EXAMPLE 4.1. Consider the formula φ and the assignment μ of Example 2.1. Suppose that MATHDPLL generates μ following the order to occurrence within φ, and that MATHSOLVE(μ) returns the conflict set (1). Thus MATHDPLL can jump back directly to the branching point $\neg(3v_1 - v_3 \leq 6)$ without exploring the right branches of $(v_3 = 3v_5 + 4)$ and $(v_1 - v_5 \leq 1)$. If instead MATHSOLVE(μ) returns the conflict set (2), then MATHSAT backtracks to $(v_3 = 3v_5 + 4)$. Thus, (2) causes no reduction in search.

When MATHSOLVE returns a conflict set η, the clause $\neg\eta$ can be added in conjunction to φ: this will prevent MATHDPLL from generating again any branch containing η. We call this technique *theory-driven learning*.

EXAMPLE 4.2. As in Example 4.1, suppose MATHSOLVE(μ) returns the conflict set (1). Then the clause $(2v_2 - v_3 > 2) \vee \neg(3v_1 - 2v_2 \leq 3) \vee (3v_1 - v_3 \leq 6)$ is added in conjunction to ϕ. Thus, whenever a branch containts two elements of (1), MATHDPLL will assign the third to false by unit propagation.

As in the Boolean case, learning must be used with some care, since it may cause an explosion in the size of φ. Therefore, some techniques can be used to discard learned clauses when necessary [8]. Notice however the difference with standard Boolean backjumping and learning [8]: in the latter case, the conflict set

propositionally falsifies the formula, while in our case it is inconsistent from the mathematically viewpoint.

4.2. THEORY-DRIVEN DEDUCTION

With early pruning, MATHSOLVE is used to check whether μ is LAL-satisfiable and thus to possibly prune whole branches of the search. It is also possible to use MATHSOLVE to reduce the remaining Boolean search: the mathematical analysis of μ performed by MATHSOLVE can discover that the value of some mathematical atom $\psi \notin \mu$ is already determined, based on some subset $\mu' \in \mu$ of the current assignment. For instance, consider the case where the literals $(v_1 = v_2)$ and $(v_2 - v_3 = 4)$ are in the current (partial) assignment μ, while $(v_1 - v_3 = 4)$ is currently unassigned. Since $\{(v_1 = v_2), (v_2 - v_3 = 4)\} \models (v_1 - v_3 = 4)$, atom $(v_1 - v_3 = 4)$ must be assigned to $\top$, because assigning it to $\bot$ would make μ LAL-inconsistent.

MATHSOLVE is therefore used to detect and suggest to the Boolean search which unassigned literals have forced values. This kind of deduction is often very useful because it can trigger new Boolean constraint propagation: the search is deepened without the need to split. Moreover, the implication clauses describing the deduction (e.g., $\neg(v_1 = v_2) \vee \neg(v_2 - v_3 = 4) \vee (v_1 - v_3 = 4)$) can be learned at the Boolean level, and added to the main formula: this constrains the remaining Boolean search even after backtracking.

4.3. A STACK-BASED INTERFACE TO MATHSOLVE

Since the search is driven by the 'stack-based' Boolean procedure, we define a stack-based interface to call the math solver. In this way, MATHSOLVE can significantly exploit previous computations. Consider the following trace (left column first, then right):

MATHSOLVE (μ_1)	$\Rightarrow$ *Sat*	Undo μ_2	
MATHSOLVE $(\mu_1 \cup \mu_2)$	$\Rightarrow$ *Sat*	MATHSOLVE $(\mu_1 \cup \mu_2')$	$\Rightarrow$ *Sat*
MATHSOLVE $(\mu_1 \cup \mu_2 \cup \mu_3)$	$\Rightarrow$ *Sat*	MATHSOLVE $(\mu_1 \cup \mu_2' \cup \mu_3')$	$\Rightarrow$ *Sat*
MATHSOLVE $(\mu_1 \cup \mu_2 \cup \mu_3 \cup \mu_4)$	$\Rightarrow$ *Unsat*	MATHSOLVE $(\mu_1 \cup \mu_2' \cup \mu_3' \cup \mu_4')$	$\Rightarrow$ *Sat*

On the left, an assignment is repeatedly extended until a conflict is found. We notice that MATHSOLVE is invoked (during early pruning calls) on *incremental* assignments. When a conflict is found, the search backtracks to a previous point (on the right), and MATHSOLVE is then restarted from a previously visited state. Based on these considerations, our MATHSOLVE is not a function call: it has a persistent state and is *incremental* and *backtrackable*. Incremental means that it avoids restarting the computation from scratch whenever it is given in input an assignment μ' such that $\mu' \supset \mu$ and μ has already proved satisfiable. Backtrackable means that it is possible to return to a previous state on the stack

in a relatively efficient manner. Therefore MATHSOLVE has primitives to *add* constraints to the current state, to *set backtrack points*, and to *jump back* to a previously set backtrack point.

4.4. FILTERING

Another way of speeding MATHSOLVE is to give it smaller but in some sense sufficient sets of constraints.

4.4.1. *Pure Literal Filtering*

Assume that a math-atom ψ occurs only positively in the formula ϕ, that is, there is no clause in ϕ having the literal $\neg\psi$. That is, ψ is a *pure literal*. Now if ψ is assigned to false in the current truth assignment μ, that is, $\neg\psi \in \mu$, we don't have to pass $\neg\psi$ to MATHSOLVE. The reason is that if an extension μ' of μ propositionally satisfies ϕ, so will $\mu'\backslash\{\neg\psi\}$ as ψ is a pure literal. Similar analysis applies to the case in which ψ occurs only negatively in ϕ.

Notice that if a pure literal ψ is assigned to true in μ, then it *has to* be passed to MATHSOLVE. Furthermore, one may *not* fix ψ to true before the MATHDPLL search as in the purely Boolean case.

4.4.2. *Theory-Deduced Literal Filtering*

Another way of reducing the number of math-atoms given to MATHSOLVE is to exploit theory-deduced clauses, i.e., those clauses resulting from theory-driven learning (Section 4.1), theory-driven deduction (Section 4.2), and static learning (Section 4.6). For each theory-deduced clause $C = l_1 \vee \ldots \vee l_n$, each l_i being a math-atom or its negation, the truth assignment $\{\neg l_1, \ldots, \neg l_n\}$ is LAL-unsatisfiable. That is, all interpretations that satisfy all $\neg l_1, \ldots, \neg l_{n-1}$ must satisfy the literal l_n. Therefore, if the current truth assignment μ contains the literals $\neg l_1, \ldots, \neg l_{n-1}$, and the literal l_n is forced to true by unit propagation on the clause C, there is no need to pass l_n to MATHSOLVE as μ is LAL-satisfiable iff $\mu \cup \{l_n\}$ is. In order to detect these cases, the theory-deduced clauses can be marked with a flag.

Combining the filtering methods requires some care. The literals $\neg l_1, \ldots, \neg l_{n-1}$ in the current truth assignment must have been passed to MATHSOLVE (i.e., not filtered) in order to apply theory-deduced literal filtering to l_n.

4.5. WEAKENED EARLY PRUNING

Early pruning calls are used only to prune the search; if the current (partial) assignment μ is found to be unsatisfiable, the search backtracks, but if it is found to be satisfiable, the search goes deeper and the assignment will be extended.

Therefore, during early pruning calls MathSolve does not have to detect *all* inconsistencies; as long as calls to MathSolve at the end of a search branch faithfully detect inconsistency, correctness and completeness are guaranteed.

We exploit this fact by using a faster, but less powerful version of MathSolve for early pruning calls. Specifically, in the $\mathbb{R}$ domain, handling disequalities requires an extra solver that is often time-consuming (see Section 5). As disequalities in $\mathbb{R}$ are typically very low-constraining, and thus very rarely cause inconsistency, during early pruning calls MathSolve ignores disequalities, which are instead considered when checking complete search branches.

In the $\mathbb{Z}$ domain, as the theory of linear arithmetic on $\mathbb{Z}$ is much harder, in theory and in practice, than that on $\mathbb{R}$ [9], during early pruning calls MathSolve looks for a solution on the reals only.

4.6. STATIC LEARNING

Before starting the actual MathDpll search, the problem can be preprocessed by adding some basic mathematical relationships among the math-atoms as Boolean constraints to the problem. As the added constraints are consequences of the underlying theory, the satisfiability of the problem is preserved. The new constraints may significant help to prune the search space in the *Boolean level*, thus avoiding some LAL-unsatisfiable models and calls to the more expensive MathSolve. In other words, before the search, we learn, at low cost, some basic facts that most often would have to be discovered, at a much higher cost, by the math solver during the search.

The simplest case of static learning is based on (in)equalities between math-terms and constants. Assume that ϕ contains a set of math-atoms of form $S_t = \{(t \bowtie_1 c_1), \ldots, (t \bowtie_n c_n)\}$, where t is a math-term $\bowtie_i \in \{<, \leq, =, \geq, >\}$, and c_i are constant. First, ϕ is conjoined with a set of constraints over the equality atoms of form $(t = c_i)$ in S_t, ensuring that at most one of them can be true. This can be achieved with pairwise mutual exclusion constraints of form $\neg(t = c_i) \vee \neg(t = c_j)$. Second, the math-atoms in S_t are connected with a linear number of binary constraints that compactly encode the obvious mathematical (in)equality relationship between them. For instance, if $S_t = \{(t \leq 2), (t = 3), (t > 5), (t \geq 7)\}$, then ϕ is conjoined with the constraints $(t = 3) \rightarrow \neg(t > 5)$, $(t = 3) \rightarrow \neg(t \leq 2)$, $(t \leq 2) \rightarrow \neg(t > 5)$, and $(t \geq 7) \rightarrow (t > 5)$. Now $(t \geq 7)$ implies $(t > 5)$, $\neg(t = 3)$ and $\neg(t \leq 2)$ in the *Boolean level*.

Furthermore, some facts among difference constraints of the form $t_1 - t_2 \bowtie c$, $\bowtie \in \{<, \leq, =, \geq, >\}$, can be easily derived and added. First, mutually exclusive pairs of difference constraints are handled. E.g., if $(t_1 - t_2 \leq 3), (t_2 - t_1 < -4) \in$ *MathAtoms* (ϕ), then the clause $\neg (t_1 - t_2 \leq 3) \vee \neg (t_2 - t_1 < -4)$ is conjoined to ϕ. Second, clauses corresponding to triangle inequalities and equalities between difference constraints are added. For example, if $(t_1 - t_2 \leq 3), (t_2 - t_3 < 5), (t_1 - t_3 < 9) \in$ *MathAtoms* (ϕ), then $(t_1 - t_2 \leq 3) \wedge t_3 < 5) \rightarrow (t_1 - t_3 < 9)$ is added

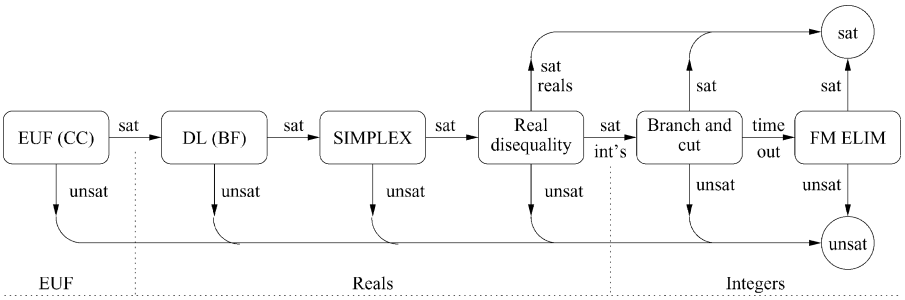

Figure 2. Control flow of MATHSOLVE.

to ϕ. Similarly, for $(t_1 - t_2 = 3)$, $(t_2 - t_3 = 0)$, $(t_1 - t_3 = 5) \in \mathit{MathAtoms}(\phi)$ we add the constraint $(t_1 - t_2 = 3)$ A $(t_2 - t_3 = 0) \rightarrow \neg(t_1 - t_3 = 5)$ to ϕ.

4.7. CLUSTERING

At the beginning of the search, $\mathit{MathAtoms}(\phi)$, that is, the set of mathematical atoms, is partitioned into a set of disjoint *clusters* $C_1 \cup \cdots \cup C_k$: intuitively, two atoms belong to the same cluster if they share a variable. If L_i is the sets of literals built with the atoms in cluster i, it is easy to see that an assignment μ is LAL-satisfiable if and only if each $\mu \cap L_i$ is LAL-satisfiable. Based on this idea, instead of having a single, monolithic solver for linear arithmetic, the mathematical solver is instantiated k different times. Each is responsible for handling the mathematical reasoning within a single cluster. A dispatcher is responsible for the activation of the suitable mathematical solver instances, depending on the mathematical atoms occurring in the assignment to be analyzed.

The advantage of this approach is manifold. First k solvers running on k disjoint problems are typically faster than running one solver monolithically on the union of the problem. Furthermore, the construction of smaller conflict sets becomes easier, and this may result in a significant gain in the overall search. Finally, when caching the results of previous calls to the linear solvers, it increases the likelihood of a hit.

5. A Layered MATHSOLVE

In this section, we discuss the structure of MATHSOLVE. We disregard the issues related to clustering, since the different instances of MATHSOLVE that result are completely independent of each other. MATHSOLVE is responsible for checking the satisfiability of a set of mathematical atoms μ and returning, as appropriate, a model or a conflict set.

In many calls to MathSolve, a general solver for linear constraints is not needed: very often, the unsatisfiability of the current assignment μ can be established in less expressive, but much easier, subtheories. Thus, MathSolve is organized in a *layered hierarchy* of solvers of increasing solving capabilities. If a higher-level solver finds a conflict, then this conflict is used to prune the search at the Boolean level; if it does not, the lower-level solvers are activated.

Layering can be understood as trying to favor faster solvers for more abstract theories over slower solvers for more general theories. Figure 2 shows a rough idea of the structure of MathSolve. Three logical components can be distinguished. First, the current assignment μ is passed to the *equational solver*, which deals only with (positive and negative) equalities (Section 5.1). Secondly, if this solver does not find a conflict, MathSolve tries to find a conflict over the reals (see Section 5.2). Third, if the current assignment is also satisfiable over the reals and the variables are to be interpreted over the integers, a solver for linear arithmetic over the integers is invoked (see Section 5.3).

5.1. EQUALITY AND UNINTERPRETED FUNCTIONS

The first layer of MathSolve is provided by the equational solver, a satisfiability checker for the logic of unconditional ground equality over uninterpreted function symbols. It is incremental and supports efficient backtracking. The solver generates conflict sets, deduces assignments for equational literals, and can provide explanations for its deductions. Thanks to the equational solver, MathSAT can be used as an efficient decision procedure for the full logic of *equality over uninterpreted function symbols* (EUF). However, in this section we focus on the way the equational solver is used to improve the performance on LAL.

The solver is based on the basic congruence closure algorithm suggested in [29]. We slightly extend the logic by allowing for *enumerated objects* and *numbers*, with the understanding that each object denotes a distinct domain element (i.e., an object is implicitly different from all the other objects and from all numbers). Similarly, different numbers are implicitly different from each other (and from all objects).

The congruence closure module internally constructs a congruence data structure that can determine whether two arbitrary terms are necessarily forced to be equal by the currently asserted constraints, and can thus be used to determine the value of (some) equational atoms. It also maintains a list of asserted *disequations* and signals unsatisfiability if either one of these or an implicit disequation is violated by the current congruence.

If two terms are equal, an auxiliary proof tree data structure allows us to extract the reason, that is, the original constraints (and just those) that forced this

equality. If a disequality constraint is violated, we can return the reason (together with the violated inequality) as a *conflict set*.

Similarly, we can perform *forward deduction*: for each unassigned equational atom, we can determine whether the two sides are already forced to be equal by the current assignment, and hence whether the atom has to be asserted as true or false. Again, we can extract the reason for this deduction and use it to represent the deduction as a learned clause on the Boolean level.

There are two ways in which the equational solver can be used: as a full solver for a purely equational cluster or as a layer in the arithmetic reasoning process. In the first case, the equational solver is associated to a cluster not involving any arithmetic at all, which contains only equation of the $v_i \bowtie v_j$, $v_i \bowtie c_j$, with $\bowtie \in \{=, \neq\}$. As stated above, the equation solver implicitly knows that syntactically different constants in $\mathcal{D}$ are semantically distinct. Hence, it provides a full solver for some clusters, avoiding the need to call an expensive linear solver on an easy problem. This can significantly improve performance, since in practical examples a purely equation cluster often is present – typical examples are the modeling of assignments in a programming language, and gate and multiplexer definitions in circuits.

In the second case, the equational solver also receives constraints involving arithmetic operators. While arithmetic functions are treated as fully uninterpreted, the equational solver has a limited interpretation of $<$ and $\leq$, knowing only that $s < t$ implies $s \neq t$, and $s = t$ implies $s \leq t$ and $\neg(s < t)$. However, all deductions and conflicts under EUF semantics are also valid under fully interpreted semantics. Thus, the efficient equational solver can be used to prune the search space. Only if the equational solver cannot deduce any new assignments and reports a tentative model, does this model need to be analyzed by lower solvers.

5.2. LINEAR ARITHMETIC OVER THE REALS

To check a given assignment μ of linear constraints for satisfiability over the reals, MATHSOLVE first considers only those constraints that are in the difference logic fragment. That is, it considers the subassignment of μ consisting of all constraints of the forms $v_i - v_j \bowtie c$ and $v_i \bowtie c$, with $\bowtie \in \{=, \neq, <, >, \leq, \geq\}$. Satisfiability checking for this subassignment is reduced to a negative-cycle detection problem in the graph whose nodes correspond to variables and whose edges correspond to the constraints. MATHSOLVE uses an incremental version of the Bellman-Ford algorithm to search for a negative-cycle and hence for a conflict. See, for instance [13], for background information. In many practical cases, for instance in bounded model-checking problems of timed automata, a sizable amount or even all of μ is in the difference logic fragment. This causes a considerable speedup, since the Bellman-Ford algorithm is much more efficient

than a general linear solver and generally generates much better (smaller) conflict sets.

If the difference logic fragment of μ turns out to be satisfiable, MathSolve checks the satisfiability of the subassignment of μ consisting of all constraints except the disequalities by means of the simplex method. MathSolve uses a variant of the simplex method, namely, the Cassowary algorithm (see [10]), that uses slack variables to efficiently allow the addition and removal of constraints and the generation of minimal conflict set.

When this also turns out to be consistent, disequalities are taken into account: the incremental and backtrackable machinery is used to check, for each disequality $\Sigma c_i v_i \neq c_j$ in μ, and separately from the other disequalities, whether it is consistent with the non-disequality constraints in μ. We do so by adding and retracting both $\Sigma c_i v_i < c_j$ and $\Sigma c_i v_i > c_j$. If one of the disequalities is inconsistent, the assignment μ is inconsistent. However, because the theory of the reals is (logically) convex, if each disequality separately is consistent, then all of μ is consistent – this follows from a dimensionality argument, basically because it is impossible to write an affine subspace A of $\mathbb{R}^k$ as a finite union of proper affine subspaces of A.

5.3. LINEAR ARITHMETIC OVER THE INTEGERS

Whenever the variables are interpreted over the reals, MathSolve is done at this point. If the variables are to be interpreted over the integers, and the problem is unsatisfiable in $\mathbb{R}$, then it also is so over $\mathbb{Z}$. When the problem is satisfiable in the reals, it is possible that it is not so in the integers. The first step carried out by MathSolve in this case is a simple form of branch-and-cut (see, e.g., [26]) that searches for solutions over the integers by tightening the constraints. The algorithm acts on the representation of the solution space constructed over the integers and makes use of the incremental and backtrackable machinery. Branch-and-cut also takes into account disequalities.

Branch-and-cut is complete only when the solution space is bounded, and there are practical cases when it can be very slow to converge. Therefore, if it does not find either an integer solution or a conflict within a small, predetermined amount of search, the current assignment is analyzed with the Fourier–Motzkin Elimination (FME) procedure. Since it is computationally expensive, FME is called only as a last resort.

6. The MathSAT System

The MathSAT system is a general solver implementing the ideas and algorithms described earlier in this paper. It also has some other features and accepts a richer input language than pure LAL, as for example, equalities over uninterpreted functions are allowed.

It is structured in three main components: (i) a preprocessor, (ii) a Boolean satisfiability solver, and (iii) the MATHSOLVE theory reasoner.

6.1. PREPROCESSOR

MATHSAT supports a rich input language, with a large variety of Boolean and arithmetic operators, including a ternary *if-then-else* construct on the term and formula level. For reasons of simplicity and efficiency, MATHDPLL, the core engine of the solver, handles a much simplified language. Reducing the rich input language to this simpler form is done by a *preprocessor* module.

The preprocessor performs some basic normalization of atoms, so that the core engine has to deal only with a restricted set of predicates. It eliminates each ternary *if-then-else* term $t = ITE(b, t_1, t_2)$ over math terms t_1 and t_2 by replacing it with a new variable v_t and adding the boolean if-then-else constraint $ITE\ (b, v_t = t_1, v_t = t_2)$ to the formula. Furthermore, it uses a standard linear-time, satisfiability preserving translation to transform the formula (including the remaining *if-then-else* on the Boolean level) into clause normal form.

6.2. BOOLEAN SOLVER

The propositional abstraction of the math-formula produced by the preprocessor is given to the Boolean satisfiability solver extended to implement the MATHDPLL algorithm described in Section 3. This solver is built upon the MINISAT solver [17], from which it inherits conflict-driven learning and back-jumping, restarts [8, 22, 37], optimized Boolean constraint propagation based on the two-watched literal scheme, and the VSIDS splitting heuristics [28]. In fact, if MATHSAT is given a purely Boolean problem, it behaves substantially like MINISAT, as MATHSOLVE is not instantiated.★ The communication with MATHSOLVE is carried out through an interface (similar to the one in [20]) that passes assigned literals, LAL-consistency queries, and back-tracking commands and receives back answers to the queries, mathematical conflict sets, and implied literals (Section 3).

The Boolean solver has been extended to handle some options relevant when dealing with math-formulas. For instance, MATHSAT inherits MINISAT's feature of periodically discarding some of the learned clauses to prevent explosion of the formula size. However, clauses generated by theory-driven learning and forward deduction mechanisms (Section 3) are never discarded, as a default option, since they may have required a lot of work in MATHSOLVE. As a second example, it is possible to initialize the VSIDS heuristics weights of literals so that either Boolean or theory atoms are preferred as splitting choices early in the MATHDPLL search.

★ In some experiments on some very big pure SAT formulas, which are not reported here, MATHSAT took on average 10–20% more time than MINISAT to solve the same instances.

6.3. MATHSOLVE

The implementation of MATHSOLVE is composed of several software modules. The equational reasoner is implemented in C/C++ and reuses some of the data structures of the theorem prover E [33] to store and process terms and atoms. The module for handling difference constraints is developed in C++. The simplex algorithm for linear arithmetic over the reals is based on the Cassowary system [5]. The branch-and-cut procedure is implemented on top of it and uses the incrementally features of Cassowary to perform the search. For the Fourier–Motzkin elimination. MATHSOLVE uses the Omega system [30].

A very important point is that MATHSAT is able to deal with infinite-precision arithmetic. To this end, the mathematical solver handles arbitrary large rational numbers by means of the GMP library [21].

7. Experimental Evaluation

In this section we report on the experiments we have carried out to evaluate the performance of our approach. The experiments were run on a bi-processor XEON 3.0 GHz machine with 4 GB of memory (test in Section 7.2), on a 4-Processor PentiumIII 700 MHz machine with 6 GB of memory (tests in Section 7.3.1), and on a bi-processor XEON 1.4 GHz machine with 2 GB of memory (tests in Section 7.3.2), all of them running Linux RedHat Enterprise. The time limit for all the experiments was set to 300 s, and the memory limit was set to 512 MB.

An executable version of MATHSAT and the source files of all the experiments performed in the paper are available at [27].

7.1. DESCRIPTION OF THE TEST CASES

The set of benchmarks we used in the experimentation, described below, involves all the suites available in the literature we are aware of. For the test on LAL, we used the following suites. The **SAL suite**, originally presented in [32], is a set of benchmarks for ground decision procedures, derived from bounded model checking of timed automata and linear hybrid systems, and from test-case generation for embedded controllers. The **RTLC suite**, provided by the authors of [31], formalizes safety properties for RTL (see [31] for a more detailed description). The **CIRC suite**, generated by ourselves, encodes the verification of certain properties for some simple circuits. The suite is composed of three kinds of benchmark, all of them being parametric in (and scaling up with) N, that is, the width of the data-path of the circuit, so that $[0..2^N - 1]$ is the range of integer variables. In the first benchmark, the modular sum of the integers is checked for

inequality against the bitwise sum of their bit decomposition. The negation of the resulting formula is therefore unsatisfiable. In the second benchmark, two identical shift-and-add multipliers and two integers a and b are given; a and the bit decomposition of b (respectively b and the bit decomposition of a) are given as input to the first (respectively, the second) multiplier and the outputs of the two multiplier are checked for inequality. The negation of the resulting formula is therefore unsatisfiable. In the third benchmark, an integer a and the bitwise decomposition of an integer b are given as input to a shift-and-add multiplier; the output of the multiplier is compared with the constant integer value p^2, p being the biggest prime number strictly smaller than 2^N. The resulting formula is satisfiable, but it has only one solution, where $a = b = p$. The **TM suite** is a set of benchmark for (temporal) metric planning, provided to us by the authors of [36] (see also [41]).

The benchmark below have been used for the comparison in Section 7.3.2 and fall into the difference logic fragment of LAL. The **DLSAT suite** is provided to us by the authors of [14] (see the paper for more detail). The suite contains two different sets of benchmark: the first set formalizes the problem of finding the optimal schedule for the job shop problem, a combinatorial optimization problem; the second set is concerned with bounded model checking of timed automata that model digital circuits with delays, and formalizes the problem of finding the maximal stabilization time for the circuits. The **SEP suite** [34] is a set of benchmarks for separation logic (i.e., difference logic) derived from symbolic simulation of several hardware designs, which is maintained by O. Strichman. The **DTP suite** [1, 38] is a set of benchmarks from the field of temporal reasoning. The set of benchmark is similar in spirit to the standard random k-CNF SAT benchmark and consists of randomly generated 2-CNF difference formulas. For our tests we have selected 60 randomly generated DTP formulas with 35 numerical variables in the "hard" satisfiability transition area.

The SAL, TM, DLSAT, and DTP suites are in the domain of reals, while the RTCL, CIRC, and SEP suites are in the domain of integers. Because of the different sources of problems within one suite, the benchmark suites cannot be straightforwardly characterized in terms of structural properties of their formulas (except for the DPT suite, in which only positive difference inequalities in the form $(x - y \leq c)$ occur). Nearly all problems contain a significant quantity of Boolean atoms (e.g., control variables in circuits, actions in planning problems, discrete variables in timed and hybrid system). Nearly all problems contain many difference inequalities in the form $(x - y \leq c)$ (e.g., time constraints in scheduling problems and in timed and hybrid systems verification problems, range constraints in RTL, circuits). Some problems, such as ATPG problems in RTLC and timed and hybrid systems in SAL, contain lots of simple equalities in the form $(x = y)$ or $(x = c)$. The problems in the CIRC suite contain complex LAL atoms with very big integer constants, like $(b = \Sigma_i\, 2^i b_i)$ or $(x \leq 2^{32})$.

7.2. EVALUATING DIFFERENT OPTIMIZATIONS

In this section we evaluate the impact of several optimizations on the overall performance of MATHSAT. The experimental evaluation has been conducted in the following way. We chose a "default" option configuration for MATHSAT, that involves theory-driven backjumping and learning, theory-driven deduction, weakened early pruning, static learning, clustering, and EQ layering (that is, using the EUF solver as described in the second case of Section 5.1).

The configuration has been tested against each of the configuration obtained by switching off (or changing) different options *one at a time* (in other words, each version that has been tested differs from the default version only with respect to one of the optimizations). Specifically, the variants we considered are respectively the default version without (weekend) early pruning, with full early pruning, without clustering, without theory-driven deduction, without static learning, and without EQ layering.

The six variants of MATHSAT have been run on the following test suites: SAL, RTLC, CIRC, TM, DLSAT, SEP, DTP.

The scatter plots of the overall results are given in Figure 3. Each plot reports the results of the evaluation on each of the options. The *X* and *Y* axes show,

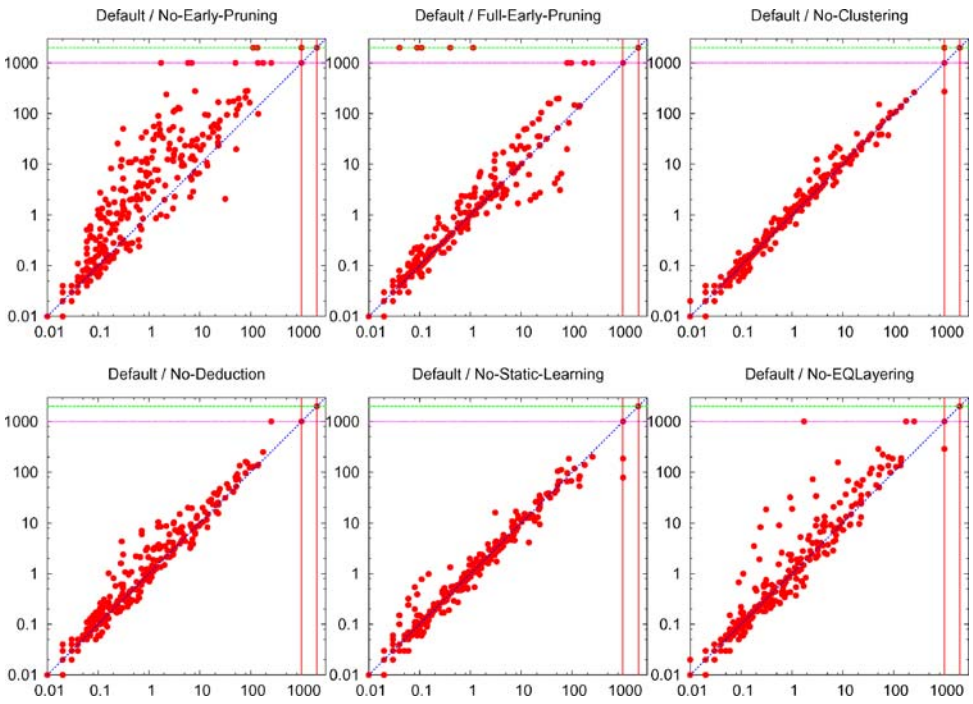

Figure 3. Scatter plots for six different variations of MATHSAT (*Y* axis), compared against the default version (*X* axis).

respectively, the performance of the default version and of the modified version of MATHSAT. A dot in the upper part of a picture, that is, above the diagonal, means that the default version performs better, and vice versa. The two uppermost horizontal lines represent benchmarks that ended in time-out (lower) or out-of-memory (higher) for the modified version of MATHSAT, whereas the two rightmost vertical lines represent time-out (left) or out-of-memory (right) for the default version. Notice that the axes are logarithmic, so that only big performance gaps are highlighted. For example, the fact that a variant is 50% faster or slower than the default on some sample (i.e., a 1.5 performance factor) is hardly discernible on these plots.

From the plots in Figure 3 we observe the following facts.

- Dropping (weakened) early pruning worsens the performances significantly, or even drastically, in most benchmarks. This is due to the fact that early pruning may allow for significant cuts to the boolean search tree, and that the extra cost of intermediate calls to MATHSOLVE is much reduced by the incrementality of MATHSOLVE. From nearly all our experiments, it turns out that early pruning causes a significant reduction of the number of branches explored in the Boolean search tree, which is proportional to the overall reduction of CPU time.
- Using full early pruning instead of its weakened version most often worsens performances, on both $\mathbb{R}$ and $\mathbb{Z}$ domains. From the experimental data, we see that full early pruning does not introduce significant reductions in the number of boolean branches explored, while the calls to MATHSOLVE require longer times on average.
  Within the $\mathbb{R}$ domian, this fact seems to suggest that ignoring disequalities in the consistency check makes MATHSOLVE faster without reducing significantly the pruning effect of the boolean search space. Within the $\mathbb{R}$ domain, this fact seems to suggest that in most cases the assignments that are consistent in $\mathbb{R}$ are consistent also in $\mathbb{Z}$ and that the overhead due to handling integers also in early pruning calls in sometimes heavy.
- Dropping clustering slightly worsens the performances in most cases, although the gaps are not dramatic. A possible explanation is that the effects of "dividing and conquering" the mathematical search space are not as relevant as those of other factors (e.g., cutting the Boolean search space). This combines with the fact that the mathematical solver is very effective in producing small conflict sets, even in presence of larger problems. In our test, only a few tests actually had more than one cluster. A more refined analysis shows that for the problems with only a single cluster the overhead is not significant.
- Dropping theory-driven deduction worsens the performance in most cases. The importance of deduction is both in the immediate effect of assigning truth values to unassigned literals, which fires Boolean constraint prop-

agation, and in the learning of extra clauses from the deduction. From our experiments, it turns out that theory-driven deduction is most effective in problems that are rich in simpler equalities like $(x = c)$ and $(x = y)$ (e.g., the problems in the RTLC and the BMC on timed system problems in SAL), which can be easily and effectively deduced by the EUF solver.

– Static learning seems to introduce only slight improvements on average. This may be due to the fact that most benchmarks derive from the encoding of verification problems, so that short clauses that can be learned easily are already part of the encodings (see, e.g., [4]). Moreover, in general, the effect of static learning is hindered in part by theory-driven learning. From our experiments, it turns out that in some benchmarks (e.g., DTP, and partly DLSAT and CIRC) where lots of clauses can be learned off-line, static-learning is effective (e.g., more than one order magnitude faster on DTP) while on other benchmarks where very few or no clause can be learned off-line, static learning is ineffective.
– Dropping EQ layering worsens the performance in most cases. We believe this is due to the fact that many practical problems contain lots of simple equalities, from which lots of information can be deduced and learned by simply applying equality propagation and congruence closure. From our experiments, it turns out that EQ layering is most effective in problems which are rich of simpler equalities like $(x = c)$ and $(x = y)$ (e.g., the problems in RTLC and the BMC on timed system problems in SAL), which can be easily and effectively handled by the EUF solver.

Figure 4 shows the impact of switching off simultaneously all six options described above. We notice that, altogether, the six optimizations improve the performances significantly, and even dramatically in most cases.

7.3. COMPARISON WITH OTHER STATE-OF-THE-ART TOOLS

In this section we report the results of the evaluation of MATHSAT with respect to other state-of-the-art tools. We distinguish the evaluation into two parts: in

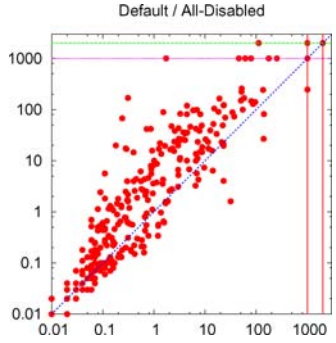

Figure 4. Scatter plots for the version of MATHSAT with all features disabled (*Y* axis) compared against the default version (*X* axis).

Section 7.3.1 we compare MATHSAT against CVC, ICS, and UCLID, which support linear arithmetic logic (LAL), whereas in Section 7.3.2 we compare MATHSAT against TSAT++ and DLSAT, which are specialized solvers for difference logic (DL).

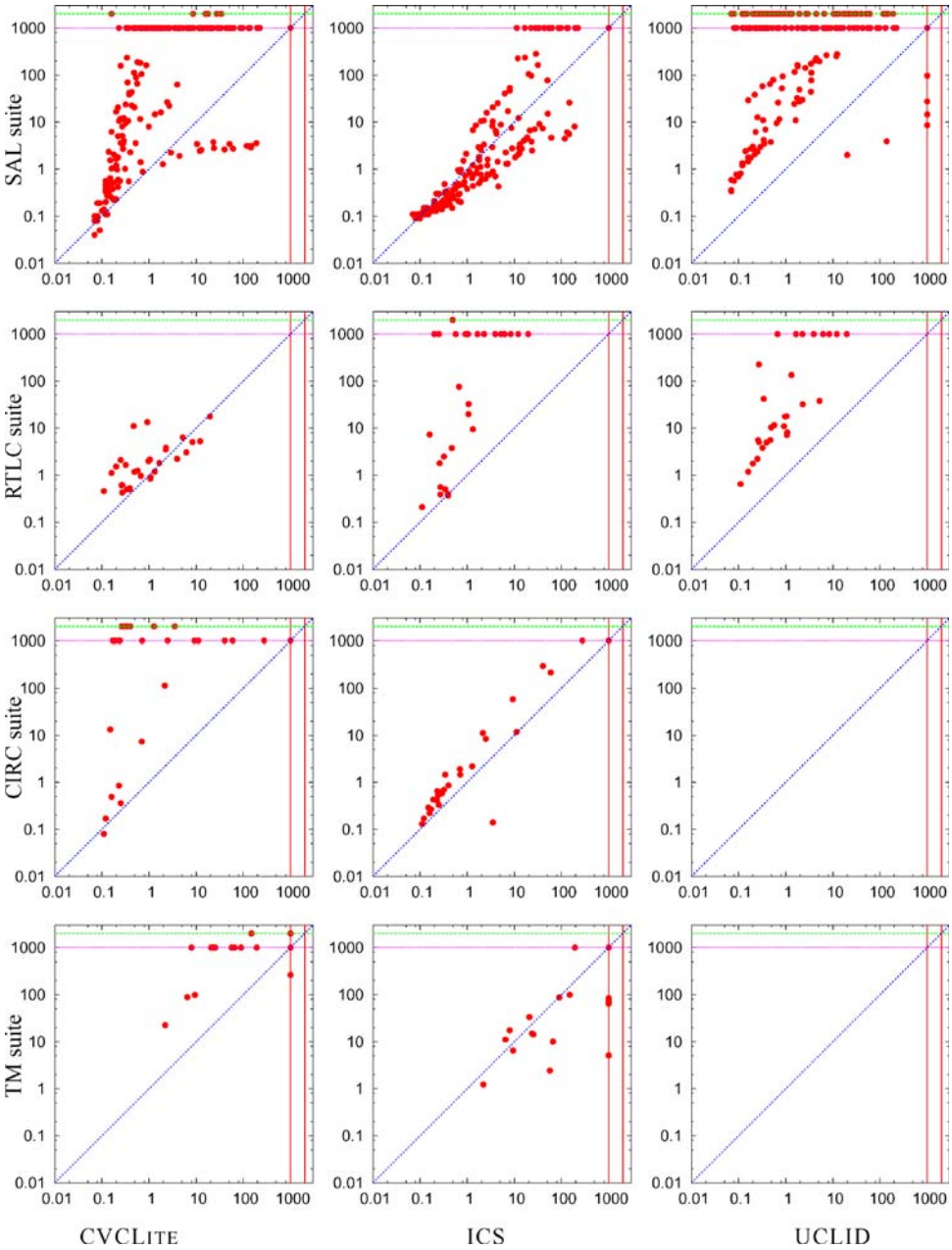

Figure 5. Execution time ratio: the *X* and *Y* axes report MATHSAT and each competitor's times, respectively.

7.3.1. *Comparison on Linear Arithmetic Logic*

We have compared MathSAT with ICS [18, 23], CVCLite [7, 15], and UCLID [35, 43]. We ran ICS version 2.0 and UCLID version 1.0. For CVCLite, we used the version available on the online repository, as of 10 October 2004, given that the latest officially released version showed a bug related to the management of integer variables (the version we used turned out to be much faster than the official one).

The results are reported in Figure 5. Each column shows the comparison between MathSAT and, respectively, CVCLite, ICS and UCLID. Each of the rows corresponds to the comparison of the four systems on the SAL, RTCL, CIRC, and TM test suites, respectively.

Each point in the scatter plot corresponds to a problem run; on the X axis we have the execution time of MathSAT, while the Y axis shows the execution time of the competitor system. A point above the diagonal means a better performance of MathSAT and vice versa. The two uppermost horizontal lines represent benchmarks that ended in time-out (lower) or out-of-memory (higher) for the competitor system, whereas the two rightmost vertical lines represent time-out (left) or out-of-memory (right) for MathSAT.

The comparison with CVCLite shows that MathSAT performs generally much better on the majority of the benchmarks in the SAL suite (CVCLite timeouts on several of them, MathSAT only on five of them). On the RTLC suite, the comparison is slightly in favor of MathSAT. For the CIRC and TM suites, the comparison is definitely in favor of MathSAT, although there are a few problems in the TM suite that neither of the systems can solve.

The comparison with ICS is reported in the second column. We see that on the SAL suite (i.e., on ICS own test suite) ICS is slightly superior on the smaller problems. However, MathSAT performs slightly better on the medium and significantly better on the most difficult problems in the suite, where ICS repeatedly times out. In the RTLC suite, ICS is clearly dominated by MathSAT. In the CIRC suite MathSAT performs better on nearly all tests, although the performance gaps are not impressive. In the TM suite, ICS performs slightly better than MathSAT.

The comparison with UCLID is limited to the problems that can be expressed, that is, some problems in SAL and RTLC, and shows a very substantial performance gap in favor of MathSAT.

An alternative view of the comparison is shown in Figures 6 and 7 (these curves are also known as *runtime distributions*). For each of the systems, we report the number of benchmarks solved (Y axis) in a given amount of time (X axis) (the samples are ordered by increasing computaion time). The upper point in the trace also shows how many samples were solved within the time limit. (Notice that the data for UCLID must be interpreted with care because it was confronted only with a subset of the problems. For the same reason, UCLID is not reported in the totals).

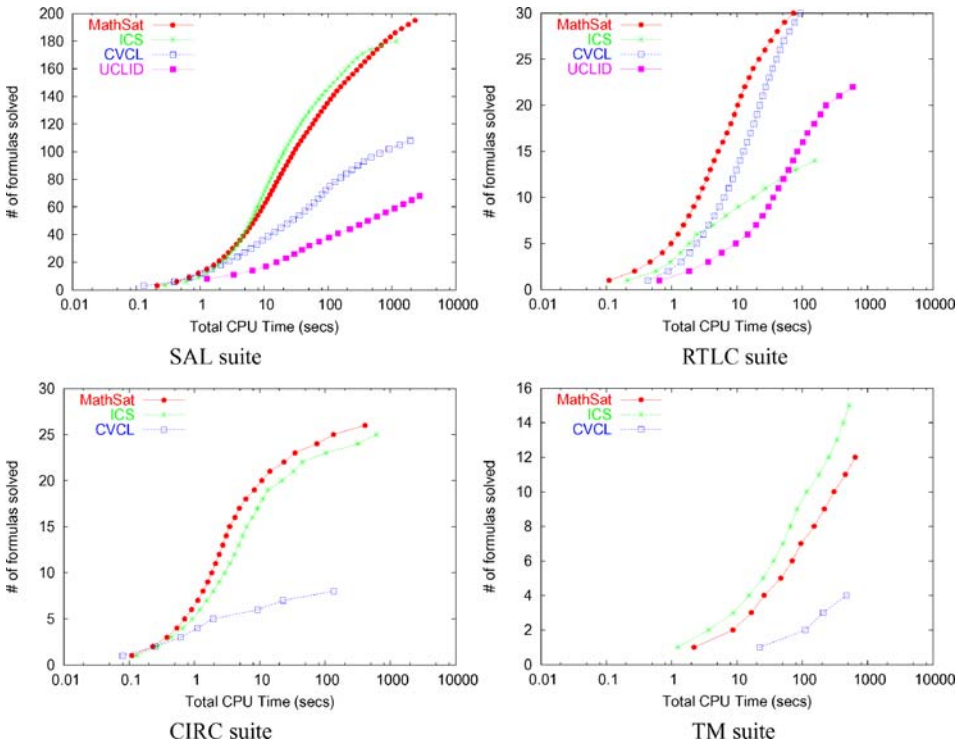

Figure 6. Number of benchmarks solved (*Y* axis) *versus* time (*X* axis) for each suite.

The curves highlight that UCLID is the worst scorer except for the RTLC suite, where it performs better than ICS, that MATHSAT and ICS perform globally better than CVCLITE, and that MATHSAT is sometimes slower on the smaller problems than ICS, but more powerful when it comes to harder problems.

One potential criticism to every empirical comparison is that the choice of the test cases may bias the results. For our tests, however, we remark that we have run *all the test cases* used by the ICS team in [16], that we have also introduced

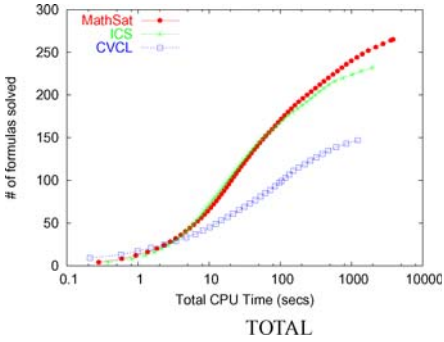

Figure 7. Number of benchmarks solved (*Y* axis) *versus* time (*X* axis) (all suites).

other suites with problems from other application domains, and that, except for the CIRC suite, all the suites we have used have been proposed by other authors in previous papers.

7.3.2. *Comparison on Difference Logic*

We also compared MATHSAT with TSAT++ [2, 42] and DLSAT [14], which are specialized solvers for difference logics. We did not include in the comparison SEP [34, 40], a decision procedure based on an eager encoding in propositional logic, since it is known to be outperformed by TSAT++ [2].

In Figure 8 we report the results of the comparison between MATHSAT and TSAT++ (left column), and DLSAT (right column). Figure 9 shows an overall comparison using runtime distributions.

MATHSAT performs slightly better than TSAT++ on the DLSAT suite, slightly worse or equivalently better on the SEP suite, and significantly better on the DTP suite (i.e., TSAT++ own suite). MATHSAT performs significantly better than DLSAT on its own suite, slightly worse on the SEP suite (notice that the samples

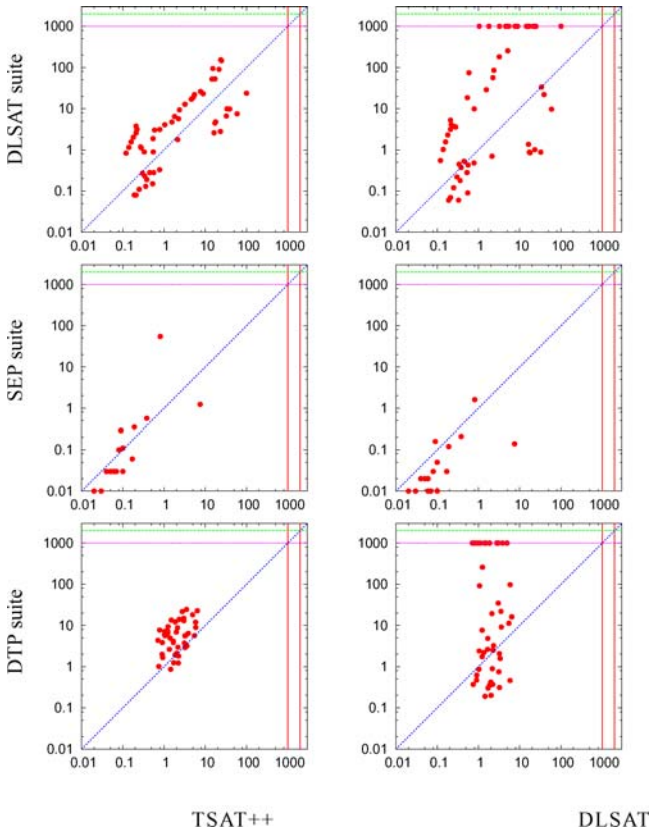

Figure 8. Execution time ratio: the *X* and *Y* axes report MATHSAT and each competitor's times, respectively.

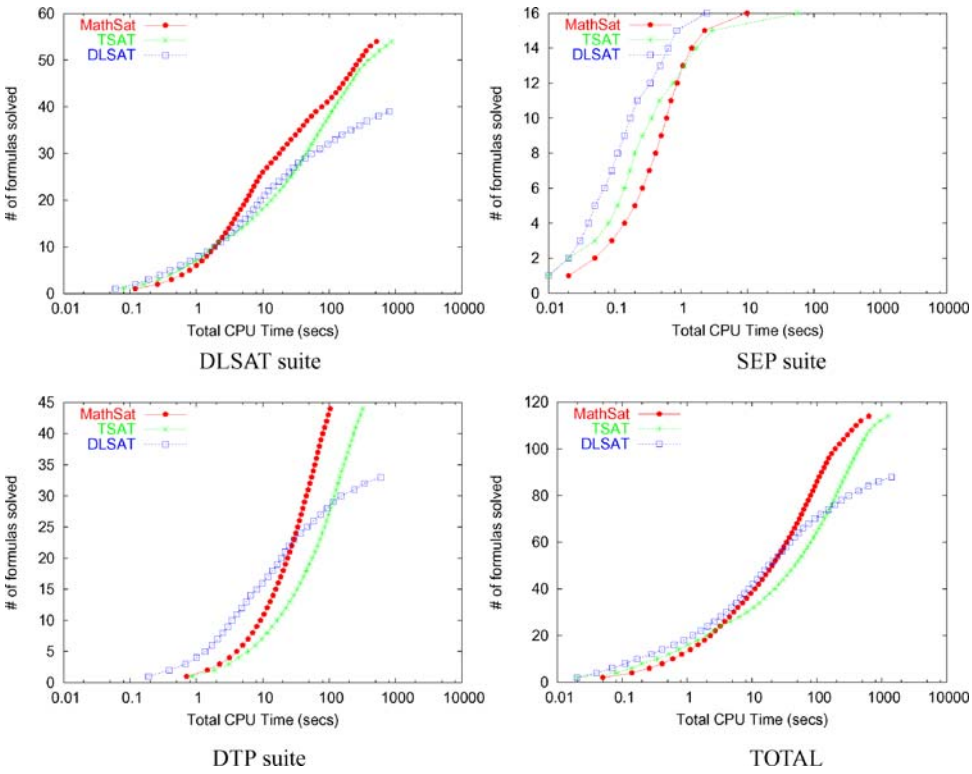

Figure 9. Number of benchmarks solved (*Y* axis) *versus* time (*X* axis) for each suite.

here are much fewer and much simpler) and significantly better in the DTP suite. On the whole, we can see that MATHSAT and TSAT++ both outperform DLSAT. Interestingly, MATHSAT exhibits on these problems a behavior that is comparable to or even better than TSAT++, which is a highly specialized solver, despite its ability to deal with a larger class of problems.

8. Related Work

In this paper we have presented a new decision procedure for linear arithmetic logic. The verification problem for LAL is well known and has received a lot of interest in the past. In particular, decision procedures are the ones considered in Section 7.3, namely, CVCLITE [7, 15], ICS [18, 23], and UCLID [35, 43].

CVCLITE is a library for checking validity of quantifier-free first-order formulas over several interpreted theories, including real and integer linear arithmetic, arrays, and uninterpreted functions. CVCLITE replaces the older tools SVC and CVC [15]. ICS is a decision procedure for the satisfiability of formulas in a quantifier-free, first-order theory containing both uninterpreted function symbols and interpreted symbol from a set of theories including arithmetic,

tuples, arrays, and bit vectors. UCLID is a tool incorporating a decision functions procedure for arithmetic of counters, the theories of uninterpreted functions and equality (EUF), separation predicates, and arrays. UCLID is based on an "eager" reduction to propositional SAT; that is, the input formula is translated into a SAT formula in a single satisfiability-preserving step, and the output formula is checked for satisfiability by a SAT solver.

In this paper, we have compared these tools using benchmarks from linear arithmetic logic (in the case of UCLID the subset of arithmetic of counters). A comparison on the benchmarks dealing with the theory of EUF is part of our future work.

Other relevant systems are Verifun [19], a tool using lazy-theorem proving based on SAT-solving, suporting domain-specific procedures for the theories of EUF, linear arithmetic and the theory of arrays, and the tool Zapato [6], a tool for counterexample-driven abstraction refinement whose overall architecture is similar to Verifun. The DPLL(T) [20] tool is a decision procedure for the theory of EUF. Similarly to MathSAT, DPLL(T) is based on a DPLL-like SAT-solver engine coupled with an efficient congruence closure module [29] that has inspired our own equational reasoner. However, our use of EUF reasoning is directed to tackling the harder problem of LAL satisfiability.

ASAP [25] is a decision procedure for quantifier-free Presburger arithmetic (that is, the theory of LAL over nonnegative integers). ASAP is implemented on top of UCLID and would have been a natural candidate for our experimental evaluation; unfortunately, a comparison was not possible because neither the system nor the benchmarks described in [25] have been made available.

We mentioned HDPLL, a decision procedure for LAL, specialized for the verification of circuits at the RTL level [31]. The procedure is based on DPLL-like Boolean search engine integrated with a constraint solver based on Fourier-Motzkin elimination and finite domain constraint propagation. According to the experimental results in [31], HDPLL seems to be very effective for its application domain. We are very interested in incorporating some of the ideas into MathSAT and in performing a thorough experimental comparison. However, HDPLL is not publicly available.

Concerning the fragment of difference logic, other related tools are the ones considered in Section 7.3.2, namely, TSAT++ [2, 42], and DLSAT [14]. While TSAT++ and DLSAT implemented an approach similar to MathSAT, they are specialized to dealing with difference logics and do not implement any form of layering. In general, TSAT++ appears to be much more efficient than DLSAT, based on a lean implementation that tightly integrates the theory solver with a state-of-the-art library for SAT. An alternative approach is implemented in SEP [34, 40], that is based on a eager approach that reduces satisfiability of the difference logic to the satisfiability of a purely propositional formula.

Concerning the very different domain of constraint logic programming, we notice that some ideas related to the mathematical solver(s) presented in this

paper (i.e., layering, stack-based interfaces, theory-deduction) are to some extent similar to those presented in [24].★

9. Conclusions and Future Work

In this paper we have presented a new approach to the satisfiability of linear arithmetic logic. The work is carried out within the (known) framework of integration between off-the-shelf SAT solvers, and specialized theory solvers. We proposed several improvements. In the top level algorithm, we exploit theory learning and deduction, theory-driven backjumping, and we adopt a stack-based interface that allows for an incremental and backtrackable implementation of the mathematical solver. We also use static learning and clustering. We heavily exploit the idea of layering: the satisfiability of theory constraints is evaluated in theories of increasing strength (equality, linear arithmetic over the reals, and linear arithmetic over the integers). The idea is to prefer less expensive solvers (for weaker theories), thus reducing the use of more expensive solvers. We carried out a thorough experimental evaluation of our approach: our MATHSAT solver is able to tackle effectively a wide class of problems, with performance comparable with and often superior to the state-of-the-art competitors, both on LAL problems and against specialized competitors on the subclass of difference logics.

As future work, we plan to enhance MATHSAT by investigating different splitting heuristics and the integration of other boolean reasoning techniques, that are complementary to DPLL. An extension of MATHSAT to nonlinear arithmetics is currently ongoing, based on the integration of computer-algebraic methods. Further extensions include the development of specialized modules to deal with memory access, bit-vector arithmetic, and the extension to the integration of EUF and LA. On the side of verification, we envisage MATHSAT as a back-end for lifting SAT-based model checking beyond the Boolean case, to the verification of sequential RTL circuits and of hybrid systems.

References

1. Armando, A., Castellini, C. and Giunchiglia, E.: SAT-based procedures for temporal reasoning, in *Proc. European Conference on Planning, CP-99*.
2. Armando, A., Castellini, C., Giunchiglia, E. and Maratea, M.: A SAT-based decision procedure for the boolean combination of difference constraints, in *Proc. Conference on Theory and Applications of Satisfiability Testing (SAT'04)*, 2004.
3. Audemard, G., Bertoli, P., Cimatti, A., Kornilowicz, A. and Sebastiani, R.: A SAT based approach for solving formulas over boolean and linear mathematical propositions, in *Proc. CADE'2002*, Vol. 2392 of *LNAI*, 2002.
4. Audemard, G., Cimatti, A., Kornilowicz, A. and Sebastiani, R.: SAT-based bounded model checking for timed systems, in *Proc. FORTE'02*, Vol. 2529 of *LNCS*, 2002.

★ We are grateful to an anonymous reviewer for pointing out this fact to us.

5. Badros, G. and Borning, A.: The Cassowary linear arithmetic constraint solving algorithm: interface and implementation. Technical Report UW-CSE-98-06-04, University of Washington, 1998.
6. Ball, T., Cook, B., Lahiri, S. and Zhang, L.: Zapato: Automatic theorem proving for predicate abstraction refinement, in *Proc. CAV'04*, Vol. 3114 of *LNCS*, 2004, pp. 457–461.
7. Barrett, C. and Berezin, S.: CVC Lite: A new implementation of the cooperating validity checker, in *Proc. CAV'04*, Vol. 3114 of *LNCS*, 2004, pp. 515–518.
8. Bayardo, Jr., R. J. and Schrag, R. C.: Using CSP look-back techniques to solve real-world SAT instances, in *Proc. AAAI/IAAI'97*, 1997, pp. 203–208.
9. Bockmayr, A. and Weispfenning, V.: Solving numerical constraints, in *Handbook of Automated Reasoning*, MIT, 2001, pp. 751–842.
10. Borning, A., Marriott, K., Stuckey, P. and Xiao, Y.: Solving linear arithmetic constraints for user interface applications, in *Proc. UIST'97*, 1997, pp. 87–96.
11. Bozzano, M., Bruttomesso, R., Cimatti, A., Junttila, T., van Rossum, P., Schulz, S. and Sebastiani, R.: An incremental and layered procedure for the satisfiability of linear arithmetic logic, in *Proc. TACAS 2005*, Vol. 3440 of *LNCS*, 2005, pp. 317–333.
12. Brinkmann, R. and Drechsler, R.: RTL-Datapath verification using integer linear programming, in *Proc. ASP-DAC 2002*, 2002, pp. 741–746.
13. Cherkassky, B. and Goldberg, A.: Negative-cycle detection algorithms, *Math. Program.* **85** (1999), 277–311.
14. Cotton, S., Asarin, E., Maler, O. and Niebert, P.: Some progress in satisfiability checking for difference logic, in *Proc. FORMATS-FTRTFT 2004*, 2004.
15. CVC. CVC, CVCLITE and SVC. http://verify.stanford.edu/{CVC, CVCL, SVC}.
16. de Moura, L. and Ruess, H.: An experimental evaluation of ground decision procedures, in R. Alur and D. Peled (eds.), *Proc. 15th Int. Conf. on Computer Aided Verification-CAV04*, Vol. 3114 of *LNCS*. Boston, Massachusetts, 2004, pp. 162–174.
17. Eén, N. and Sörensson, N.: An extensible SAT-solver, in *Theory and Applications of Satisfiability Testing (SAT 2003)*, Vol. 2919 of *LNCS*, 2004, pp. 502–518.
18. Filliâtre, J.-C., Owre, S., Ruess, H. and Shankar, N.: ICS: Integrated canonizer and solver, in *Proc. CAV'01*, Vol. 2102 of *LNCS*, 2001, pp. 246–249.
19. Flanagan, C., Joshi, R., Ou, X. and Saxe, J.: Theorem proving using lazy proof explication, in *Proc. CAV'03*, Vol. 2725 of *LNCS*, 2003, pp. 355–367.
20. Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A. and Tinelli, C.: DPLL(T): fast decision procedures, in *Proc. CAV'04*, Vol. 3114 of *LNCS*, 2004, pp. 175–188.
21. GMP. GNU Multi Precision Library. http://www.swox.com/gmp.
22. Gomes, C., Selman, B. and Kautz, H.: Boosting combination search through randomization, in *Proc. of the Fifteenth National Conf. on Artificial Intelligence*, 1998, pp. 431–437.
23. ICS. ICS. http://www.icansolve.com.
24. Jaffar, J., Michaylov, S., Stuckey, P.J. and Yap, R.H.C.: The CLP(R) languages and systems, *ACM Trans. Program. Lang. Syst. (TOPLAS)* **14**(3) (1992), 339–395.
25. Kroening, D., Ouaknine, J., Seshia, S. and Strichman, O.: Abstraction-based satisfiability solving of Presburger arithmetic, in *Proc. CAV'04*, Vol. 3114 of *LNCS*, 2004, pp. 308–320.
26. Land, H. and Doig, A.: An automatic method for solving discrete programming problems, *Econometrica* **28** (1960), 497–520.
27. MATHSAT. MATHSAT. http://mathsat.itc.it.
28. Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L. and Malik, S.: Chaff engineering an efficient SAT solver, in *Proc. DAC'01*, 2001, pp. 530–535.
29. Nieuwenhuis, R. and Oliveras, A.: Congruence closure with integer offset, in *Proc. 10th LPAR*, 2003, pp. 77–89.
30. Omega. Omega. http://www.cs.umd.edu/projects/omega.

31. Parthasarathy, G., Iyer, M., Cheng, K.-T. and Wang, L.-C.: An efficient finite-domain constraint solver for circuits, in *Proc. DAC'04*, 2004, pp. 212–217.
32. SAL. SAL Suite. http://www.csl.sri.com/users/demoura/gdp-benchmark.html.
33. Schulz, S.: E-A Brainiac theorem prover, *AI Commun.* **15**(2/3) (2002), 111–126.
34. SEP. SEP Suite, http://iew3.technion.ac.il/~ofers/smtlib-local/benchmarks.html.
35. Seshia, S., Lahiri, S. and Bryant, R.: A hybrid SAT-based decision procedure for separation logic with uninterpreted function, in *Proc. DAC'03*, pp. 425–430.
36. Shin, J.-A. and Davis, E.: Continuous time in a SAT-based planner, in *Proc. AAAI-04*, 2004, pp. 531–536.
37. Silva, J. P. M. and Sakallah, K. A.: GRASP – A new search algorithm for satisfiability, in *Proc. ICCAD'96*, 1996, pp. 220–227.
38. Stergiou, K. and Koubarakis, M.: Backtracking algorithms for disjunctions of temporal constraints, *Artif. Intell.* **120**(1) (2000), 81–117.
39. Strichman, O.: On solving presburger and linear arithmetic with SAT, in *Proc. of Formal Methods in Computer-Aided Design (FMCAD 2002),* 2002.
40. Strichman, O., Seshia, S., Bryant, R.: Deciding separation formulas with SAT, in *Proc. of Computer Aided Verification, (CAV'02).*
41. TM. TM-LPSAT. http://csl.cs.nyu.edu/~jiae/.
42. TSAT. TSAT++. http://www.ai.dist.unige.it/Tsat.
43. UCLID.UCLID. http://www-2.cs.cmu.edu/~uclid.
44. Zhang, L. and Malik, S.: The quest for efficient boolean satisfiability solves, in *Proc. CAV'02*, 2002, pp. 17–36.