An Incremental and Layered Procedure for the Satisfiability of Linear Arithmetic Logic *

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
{bozzano,bruttomesso,cimatti,vanrossum}@itc.it
[2] Helsinki University of Technology, P.O.Box 5400, FI-02015 TKK, Finland
Tommi.Junttila@tkk.fi
[3] University of Verona, Strada le Grazie 15, 37134 Verona, Italy
schulz@eprover.org
[4] Università di Trento, Via Sommarive 14, 38050 Povo, Trento, Italy
rseba@dit.unitn.it

Abstract. In this paper we present a new decision procedure for the satisfiability of Linear Arithmetic Logic (LAL), i.e. boolean combinations of propositional variables and linear constraints over numerical variables. Our approach is based on the well known integration of a propositional SAT procedure with theory deciders, enhanced in the following ways.

First, our procedure relies on an *incremental* solver for linear arithmetic, that is able to exploit the fact that it is repeatedly called to analyze sequences of increasingly large sets of constraints. Reasoning in the theory of LA interacts with the boolean top level by means of a stack-based interface, that enables the top level to add constraints, set points of backtracking, and backjump, without restarting the procedure from scratch at every call. Sets of inconsistent constraints are found and used to drive backjumping and learning at the boolean level, and theory atoms that are consequences of the current partial assignment are inferred.

Second, the solver is *layered*: a satisfying assignment is constructed by reasoning at different levels of abstractions (logic of equality, real values, and integer solutions). Cheaper, more abstract solvers are called first, and unsatisfiability at higher levels is used to prune the search. In addition, theory reasoning is partitioned in different clusters, and tightly integrated with boolean reasoning.

We demonstrate the effectiveness of our approach by means of a thorough experimental evaluation: our approach is competitive with and often superior to several state-of-the-art decision procedures.

* This work has been 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. It has also been partly supported by ESACS, an European sponsored project, contract no. G4RD-CT-2000-00361, 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'Universit e della Ricerca and the University of Verona. R. Sebastiani is also sponsored by a MIUR COFIN02 project, code 2002097822_003.

1 Motivations and Goals

Many practical domains require a degree of expressiveness beyond propositional 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 Register Transfer Level, even though expressible via booleanization, might be easier to analyze at a higher level of abstraction (see e.g. [15]). Many of the verification problems arising in such domains can be naturally modeled as satisfiability in Linear Arithmetic Logic (LAL), i.e., the boolean combination of propositional variables and linear constraints over numerical variables. For its practical relevance, LAL has been devoted a lot of interest, and several decision procedures exist that are able to deal with it (e.g., SVC [17], ICS [24, 19], CVCLITE [17, 10], UCLID [36, 33], HDPLL [30]).

In this paper, we propose a new decision procedure for the satisfiability of LAL, both for the real-valued and integer-valued case. We start from a well known approach, previously applied in MATHSAT [26, 4] and in several other systems [24, 19, 17, 10, 35, 3, 21]: 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 respect to the theory.

In this paper, we extend the MATHSAT approach in the following directions. First, the linear arithmetic solver is *incremental*: since the theory solver is called to analyze increasingly large sets of constraints, theory reasoning interacts with the boolean top level by means of a stack-based interface, that enables the top level to add constraints, set points of backtracking, and backjump. In addition, sets of inconsistent constraints are identified and used to drive backjumping and learning at the boolean level, and theory atoms that are consequences of the current partial assignment are automatically inferred. Second, we make aggressive use of *layering*: a satisfying assignment is incrementally constructed by reasoning at different levels of abstractions (logic of equality, real values, and integer solutions). Cheaper, more abstract solvers are called first, and unsatisfiability at higher levels is used to prune the search. In addition, theory reasoning is partitioned in different *clusters*, and tightly integrated with boolean reasoning.

We evaluated our approach by means of a thorough experimental comparison: the MATHSAT solver is compared against the state-of-the-art systems ICS, CVCLITE, and UCLID [33] on a large set of benchmarks proposed in the literature. We show that our approach is able to deal effectively with a wide class of problems, with performances comparable with and often superior to the other systems.

This paper is structured as follows. In Sect. 2 we define Linear Arithmetic Logic. In Sect. 3 we describe the basic algorithm, the interplay between boolean and theory reasoning, and the incrementality of the theory solver. In Sect. 4 we describe the internal structure of the solver, focusing on the ideas of layering and clustering. In Sect. 5 we describe the MATHSAT system, and in Sect. 6 we present the result of the experimental evaluation. In Sect. 7 we discuss some related work; finally, in Sect. 8 we draw some conclusions and outline the directions for future work.

2 Background

Let $\mathbb{B} := \{\bot, \top\}$ be the domain of boolean values. Let $\mathcal{D}$ be the domain of either real numbers $\mathbb{R}$ or integers $\mathbb{Z}$. By *math-terms* and *math-formulas* on $\mathcal{D}$ we denote respectively the quantifier-free linear mathematical expressions and formulas built on constants, variables and arithmetical operators over $\mathcal{D}$ and on boolean propositions, closed on boolean connectives. Math-terms are either constants $c_i \in \mathcal{D}$, or variables v_i over $\mathcal{D}$, possibly with coefficients (i.e. $c_i \cdot v_j$), or applications of the arithmetic operators $+$ and $-$ to math-terms. Atomic math-formulas are either boolean propositions A_i over $\mathbb{B}$, or applications of the arithmetic relations $=, \neq, >, <, \geq, \leq$ to math-terms. Such formulas are also called *atoms*. Math formulas are either atoms or combinations of math formulas by means of 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 $\mathbb{Z}$; an atom is called *boolean* if it is a boolean proposition, otherwise it is called a *mathematical* atom. 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 mean ψ rather than $\neg\neg\psi$. We denote by $Atoms(\phi)$ the set of mathematical atoms of a math-formula ϕ.

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 $\mathcal{B}2\mathcal{M}$ (for "Boolean-to-Math") is respectively called *refinement*.

An *interpretation* in $\mathcal{D}$ is a map I which assigns values in $\mathcal{D}$ to math-terms and truth values in $\mathbb{B}$ to math-formulas, and interprets mathematical constants, arithmetical and boolean operators according to the usual semantics of arithmetical and logical symbols. We say that I *satisfies* a math-formula ϕ, written $I \models \phi$, iff $I(\phi)$ evaluates to true. E.g., 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.

We address the problem of checking the satisfiability of math-formulas. As standard boolean formulas are a strict sub-case of math-formulas, it follows trivially that the problem is NP-hard. Thus the problem is theoretically "at least as hard" as standard boolean satisfiability, and much harder in practice.

A total (resp. partial) *truth assignment* for a math-formula ϕ is a truth value assignment μ to all (resp. a subset of) the atoms of ϕ. We represent truth assignments as set of literals $\mu = \{\alpha_1, \ldots, \alpha_N, \neg\beta_1, \ldots, \neg\beta_M, A_1, \ldots, A_R, \neg A_{R+1}, \ldots, \neg A_S\}$, $\alpha_1, \ldots, \alpha_N, \beta_1, \ldots, \beta_M$ being mathematical atoms and $A_1, \ldots, A_S$ being boolean atoms, with the intended meaning that positive and negative literals represent atoms assigned to true and to false respectively.

We say that μ *propositionally satisfies* ϕ, written $\mu \models_p \phi$, iff $\mathcal{M}2\mathcal{B}(\mu) \models \mathcal{M}2\mathcal{B}(\phi)$. Intuitively, if we see a math-formula ϕ as a propositional formula in its atoms, then $\models_p$ is the standard satisfiability in propositional logic.

We say that an interpretation I satisfies μ iff I satisfies all the elements of μ. For instance, the 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 $I(v_2) = 1$. We say that an assignment or a math-formula is *LAL-satisfiable* if there is an interpretation I satisfying if, *LAL-unsatisfiable* otherwise.

Example 1. Consider the following math-formula φ:

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

The truth assignment given by the underlined literals above is:

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

μ 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

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

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

do not have any satisfying interpretation. $\diamond$

Given a LAL-unsatisfiable assignment μ, we call a *conflict set* any LAL-unsatisfiable sub-assignment $\mu' \subseteq \mu$; we say that μ' is a *minimal* conflict set if all subsets of μ' are LAL-satisfiable. E.g., both (1) and (2) are minimal conflict sets of μ.

3 The Top Level Algorithm: Boolean+Theory Solving

This section describes the MATHSAT algorithm [4] (see Fig. 1), and its extensions. MATHSAT takes as input a math-formula ϕ, and returns $\top$ if ϕ is LAL-satisfiable (with I containing a satisfying interpretation), and $\perp$ otherwise. (Without loss of generality, ϕ is assumed to be in conjunctive normal form (CNF).) MATHSAT invokes MATHDPLL on the boolean formula $\varphi := \mathcal{M}2\mathcal{B}(\phi)$. (Both $\mathcal{M}2\mathcal{B}$ and $\mathcal{B}2\mathcal{M}$ can be implemented so that they require constant time in mapping one atom.)

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

base If $\varphi == \top$, then μ propositionally satisfies $\mathcal{M}2\mathcal{B}(\phi)$. In order to check if 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 $\varphi == \perp$, then μ has lead to a propositional contradiction. Therefore MATHDPLL returns *Unsat* and backtracks.

```
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);
```

Fig. 1. High level view of the MATHSAT algorithm

unit If a literal l occurs in φ as a unit clause, then l must be assigned a true value. Thus, MATHDPLL is invoked recursively with *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.

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 literal 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 arguments *assign*$(\neg l,\varphi)$ and $\mu \cup \{\neg l\}$.

The schema of Fig. 1 is over-simplified for explanatory purposes. However, it can be easily adapted to exploit advanced SAT solving techniques (see [38] for an overview). In the rest of this section, we will focus on the interaction between boolean reasoning (carried out by MATHDPLL) and theory reasoning (carried out by MATHSOLVE) instead of on the details underlying the boolean search.

Theory-driven backjumping and learning [23, 37]. 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 conflict set generated.

Example 2. Consider the formula φ and the assignment μ of Ex. 1. Suppose that MATHDPLL generates μ following the order of 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. $\diamond$

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 3. As in Ex. 2, 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 contains two elements of (1), then MATHDPLL will assign the third to false by unit propagation. $\diamond$

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 [11]. Notice however the difference with standard boolean backjumping and learning [11]: in the latter case, the conflict set propositionally falsifies the formula, while in our case it is inconsistent from the mathematical viewpoint.

Theory-driven deduction [2, 4, 21]. With early pruning, MATHSOLVE is used to check if μ is LAL-satisfiable, and possibly close whole branches of the search. It is also possible to use MATHSOLVE to reduce the remaining boolean search: in fact, the mathematical analysis of μ performed by MATHSOLVE can allow for discovering that the value of some mathematical atoms $\psi \notin \mu$ is already determined, based on some subset $\mu' \in \mu$ being part of the current assignment. For instance, consider the case where the literals $(v_1 - v_2 \leq 4)$ and $(v_2 = v_3)$ are in the current (partial) assignment μ, while $(v_1 - v_3 \leq 5)$ is currently unassigned. Since $\{(v_1 - v_2 \leq 4), (v_2 = v_3)\} \models (v_1 - v_3 \leq 5)$, atom $(v_1 - v_3 \leq 5)$ can not be assigned to $\perp$, since this 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, since it can trigger new boolean constraint propagation: the search is deepened without the need to split. Moreover, the implication clauses (e.g. $\neg(v_1 - v_2 \leq 4) \vee \neg(v_2 = v_3) \vee (v_1 - v_3 \leq 5)$) can be learned and added to the main formula: this constrains the remaining boolean search in the event of backtracking.

Incremental and backtrackable theory solver [5, 17, 24]. Given the stack-based nature of the boolean search, the MATHSOLVE can significantly exploit previous computations. Consider the following trace (left column, then right):

MATHSOLVE (μ_1)	$\Longrightarrow$ *Sat*	Undo μ_2	
MATHSOLVE $(\mu_1 \cup \mu_2)$	$\Longrightarrow$ *Sat*	MATHSOLVE $(\mu_1 \cup \mu_2')$	$\Longrightarrow$ *Sat*
MATHSOLVE $(\mu_1 \cup \mu_2 \cup \mu_3)$	$\Longrightarrow$ *Sat*	MATHSOLVE $(\mu_1 \cup \mu_2' \cup \mu_3')$	$\Longrightarrow$ *Sat*
MATHSOLVE $(\mu_1 \cup \mu_2 \cup \mu_3 \cup \mu_4)$	$\Longrightarrow$ *Unsat*	MATHSOLVE $(\mu_1 \cup \mu_2' \cup \mu_3' \cup \mu_4')$	$\Longrightarrow$ *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 restarting 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 as input an assignment μ' such that $\mu' \supset \mu$ and μ has already been proved satisfiable. Backtrackable means that it is possible to return to a previous state on the stack in a relatively efficient manner. In fact, MATHSOLVE mimics the stack based behaviour of the boolean search.

4 Clustering and Layering in MATHSOLVE

In this section, we discuss how to optimize MATHSOLVE, based on two main ideas: *clustering* and *layering*.

Clustering At the beginning of the search, the set $Atoms(\phi)$ of all atoms occurring in the formula is partitioned into disjoint *clusters*: intuitively, two atoms (literals) belong to the same cluster if they share a variable. Say $Lits(\phi) = L_1 \cup \cdots \cup L_k$ is the so-obtained static partitioning of the literals. Because no two L_i have a variable in common, the assignment μ is satisfiable if and only if each $\mu \cap L_i$ is satisfiable.

Based on this idea, instead of having a single, monolithic solver for linear arithmetic, k different solvers are constructed, each responsible for the handling of a single cluster. The advantage of this approach is not only that running k solvers on k disjoint problems is faster then running one solver on the union of those k problems, but also a significant gain is obtained by the potential construction of smaller conflict sets. Additionally, we are hashing the results of calls to the linear solvers; if there are more linear solvers, then the likelihood of a hit increases.

Layering 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, sub-theories. 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.

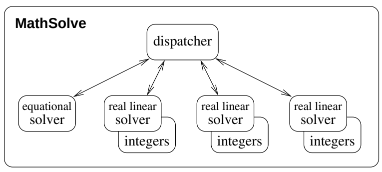


Fig. 2. Clustering and layering

Layering can be explained as trying to privilege faster solvers for more abstract theories over slower solvers for more general theories. Fig. 2 shows a rough idea of the structure of MATHSOLVE, and highlights the two places in MATHSOLVE where this layering is taking place. Firstly, the current assignment μ is passed to the *equational solver*, described in more detail in Sect. 4.1, that only deals with (positive and negative) equalities. Only if this solver does not find a conflict is a full-blown solver for linear arithmetic invoked. Secondly, the *solver for linear arithmetic*, described in Sect. 4.2, is itself layered: when reasoning about integer variables, it first tries to find a conflict over the real numbers, and looks for a conflict over the integers only in case of satisfiability.

4.1 The Equational Satisfiability Procedure

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 congruence closure algorithm suggested in [28], and reuses some of the data structures of the theorem prover E [32] to store and process terms and atoms. It internally constructs a congruence data structure that can determine if 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 one of these is violated by the current congruence. Similarly, the solver implicitly knows that syntactically different constants in $\mathcal{D}$ are semantically distinct, and efficiently detects and signals if a new equation forces the identification of distinct domain elements.

If two terms are equal, an auxiliary proof tree data structure allows us to extract the reason, i.e. 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 if 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 solver for equational clusters, or as a layer in the arithmetic reasoning process. In the first case, only those clusters not involving any arithmetic at all are given to the equational solver: the dispatcher moves to the equational solver only equations of the form $v_i \bowtie v_j$, $v_i \bowtie c_j$, with $\bowtie \in \{=, \neq\}$. Thus, the equational solver 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 it is often the case that a purely equational cluster is present – typical examples are modeling of assignments in a programming language, and gate and multiplexer definitions in circuits.

In the second case, the dispatcher also passes constraints involving arithmetic operators to the equational solver. 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,

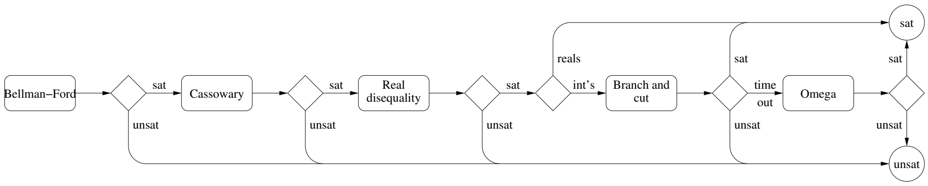


Fig. 3. Control flow of linear solver

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, this model has to be verified (or rejected) by lower level solvers.

4.2 The Solver for Linear Arithmetic

The task of the linear solver is to check a given assignment μ of linear constraints ($\sum_i c_i v_i \bowtie c_j$, with $\bowtie \in \{=, \neq, >, <, \geq, \leq\}$) for satisfiability and, as appropriate, return a model or a conflict set.

The linear solver itself is also layered, running faster, more general solvers first and using slower, more specialized solvers only if the early ones do not detect an inconsistency. The control flow through the linear solver is given in Fig. 3.

First, we consider only those constraints that are in the difference logic fragment, i.e., 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 edge correspond to the constraints. We use an incremental version of the Bellman-Ford algorithm to search for a negative-cycle and hence for a conflict [16].

Second, we try to determine if the current assignment μ is consistent over the reals, by means of the Cassowary constraint solver. Cassowary [13, 8] is a simplex-based solver over the reals, using slack variables to efficiently allow the addition and removal of constraints and the generation of a minimal conflict set.

Cassowary is called on μ minus the disequalities (i.e., with $\bowtie$ equal to $\neq$). When Cassowary does not find a conflict, its incremental and backtrackable machinery is used to check for each disequality $\sum c_i v_i \neq c_j$ in μ separately if it is consistent with the non-disequality constraints in μ. We do so by adding and retracting both $\sum c_i v_i < c_j$ and $\sum c_i v_i > c_j$. Of course, if one of the disequalities is inconsistent, the whole assignment μ is inconsistent. However, if each disequality separately is consistent, then by dimensionality reasons all of μ is consistent. [1]

Whenever the variables are interpreted over the reals, MATHSOLVE is done at this point. If the variables are interpreted over the integers, and the problem is unsatisfiable in the reals, then it is so in the integers. When the problem is satisfiable in the reals, a simple form of branch-and-cut is carried out, to search for solutions over the integers, using Cassowary's incremental and backtrackable machinery. If branch-and-cut does

[1] 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.

not find either an integer solution or a conflict within a small, predetermined amount of search, the Omega constraint solver [29] is called on the current assignment. Omega is a constraint solver over the integers based on Fourier-Motzkin. Since it is computationally expensive to call, does not have an incremental and backtrackable interface and also is not capable of generating a conflict set, it is called only as a last resort.

One implementation issue is that everything has to be done with infinite precision. For this, we modified the Cassowary solver to handle arbitrary large rational numbers.

5 The MATHSAT System

The actual MATHSAT system has three components: (i) a preprocessor, (ii) a boolean satisfiability solver, and (iii) the MATHSOLVE theory solver described in Sect. 4.

Preprocessor MATHSAT allows the input formulas to contain constructions that cannot be handled directly by the MATHDPLL algorithm. These features and some optimizations are handled by a *preprocessor*. First, MATHSAT allows the input formulas to be in non-clausal form and to include boolean operators such as $\rightarrow$ and ternary if-then-else. Thus the last step in the preprocessor is to translate the formula into CNF by using a standard linear-time satisfiability preserving translation. Second, the input formulas may contain uninterpreted functions and predicates. If they are used in a mixed way that cannot be handled either by the EUF solver or linear arithmetic solver alone (e.g. an atom $f(x) + f(z) = c$), the preprocessor uses Ackermann's reduction to eliminate them [1].

In addition, the preprocessor uses a form of *static learning* to add some satisfiability preserving constraints that help to prune the search space in the boolean level. For instance, if a formula ϕ contains a set of math-atoms of form $\{(t = c_1), ..., (t = c_n)\}$, where t is a math-term and c_i are mutually disjoint constants, then ϕ is conjuncted with constraints enforcing that at most one of the atoms can be true. Similarly, a linear number of constraints encoding the basic mathematical relationships between simple (in)equalities of the form $t \bowtie c_i$, $\bowtie \in \{<, \leq, = \geq, >\}$, are added. E.g. if $(t \leq 2), (t = 3), (t > 5)$ are math-atoms in ϕ, then ϕ is conjuncted with the constraints $(t = 3) \rightarrow \neg(t > 5)$, $(t = 3) \rightarrow \neg(t \leq 2)$, and $(t \leq 2) \rightarrow \neg(t > 5)$. Furthermore, some facts between difference constraints of form $t_1 - t_2 \bowtie c$, where $\bowtie \in \{<, \leq, \geq, >\}$ and c is a constant, are included: (i) mutual exclusion of conflicting constraints is forced, e.g. for $(t_1 - t_2 \leq 3)$ and $(t_2 - t_1 < -4)$, the constraint $\neg(t_1 - t_2 \leq 3) \vee \neg(t_2 - t_1 < -4)$ is added, and (ii) constraints corresponding to triangle inequalities are added, e.g. for $(t_1 - t_2 \leq 3)$, $(t_2 - t_3 < 5)$, and $(t_1 - t_3 < 9)$, the constraint $(t_1 - t_2 \leq 3) \wedge (t_2 - t_3 < 5) \rightarrow (t_1 - t_3 < 9)$ is included.

Boolean solver The math-formula in CNF produced by the preprocessor is given to the boolean satisfiability solver extended to implement the MATHDPLL algorithm in Sect. 3. In MATHSAT, the boolean solver is built upon the MINISAT solver [18]. Thus it inherits conflict-driven learning and back-jumping, restarts [34, 11, 22], optimized boolean constraint propagation based on the two-watched literal scheme, and an effective splitting heuristics VSIDS [27] for free. It communicates with MATHSOLVE through an interface (resembling the one in [21]) that passes assigned literals, LAL-consistency queries and backtracking commands to MATHSOLVE and gets back answers to the queries, mathematical conflict sets and implied literals (Sect. 3). The boolean solver is also extended to handle some optimization 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. But because clauses generated by theory-driven learning and forward deduction mechanisms (Sect. 3) may have required a lot of work in MATHSOLVE, as a default option they are never discarded. As a second example, it is possible to initialize the VSIDS heuristics weights of literals so that either boolean or mathematical atoms are preferred as splitting choices early in the MATHDPLL search.

Furthermore, as the theory of linear arithmetic on $\mathbb{Z}$ is much harder, in theory and in practice, than that on $\mathbb{R}$ [12], in early pruning calls we only use weaker but faster versions of MATHSOLVE, which look for a solution on the reals only. This is based on the heuristic consideration that, in practice, if an assignment is consistent in $\mathbb{R}$ it is often also consistent in $\mathbb{Z}$, and that early pruning checks are not necessary for the correctness and completeness of the procedure.

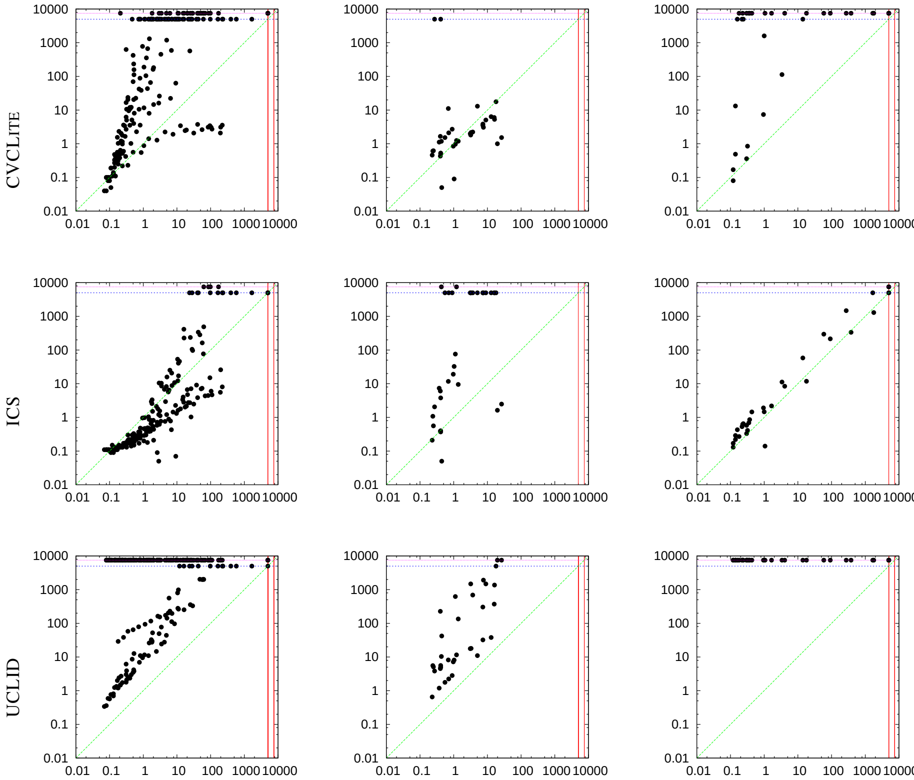
6 Experimental Evaluation

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 4-processor PentiumIII 700 MhZ machine with more than 6 Gb of memory, running Linux RedHat 7.1. An executable version of MATHSAT and the source files of all the experiments performed in the paper are available at [26].

Description of the test cases The first set of experiments was performed on the SAL suite [31], a set of benchmarks for ground decision procedures. The suite is derived from bounded model checking of timed automata and linear hybrid systems, and from test-case generation for embedded controllers. The problems are represented in non-clausal form, and constraints are in linear arithmetic. This suite contains 217 problems, 110 of which are in the separation logic fragment.

The second set of experiments was performed on a benchmark suite (called RTLC hereafter) formalizing safety properties for RTL circuits, provided to us by the authors of [30] (see [30] for a more detailed description of the benchmarks).

Finally, we used a benchmark suite (CIRC) generated by ourselves, verifying properties for some simple circuits. The suite is composed of three different kinds of benchmarks, all of them being parametric in (and scaling up with) N, where $[0..2^N - 1]$ is the range of an integer variable. In the first benchmark, the modular sum of two integers is checked for equality against the bit-wise 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 multipliers are



SAL suite RTLC suite CIRC suite

Fig. 4. Execution time ratio: the X and Y axes report MATHSAT and each competitor's times respectively.

checked for equality. 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: $a = b = p$ and corresponding bit values.

Comparison with other state-of-the-art tools We evaluated the performance of MATHSAT with respect to other state-of-the-art tools, namely ICS, CVCLITE and UCLID. For ICS and UCLID, the latest officially released versions were used for the comparative evaluation. For CVCLITE, we used the latest available version on the on-line repository, given that the latest officially released version showed a bug related to the management of integer variables. Moreover, the version we used turned out to be much faster than the other one. The time limit for these experiments was set to 1800

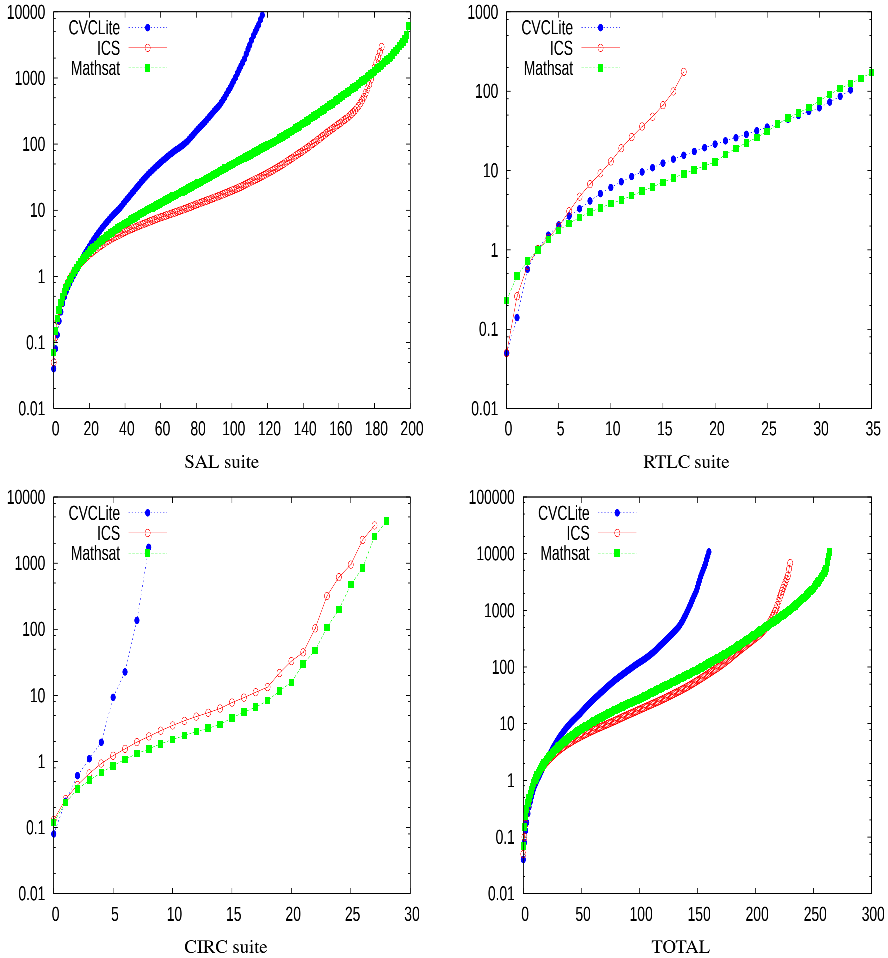


Fig. 5. Number of benchmarks solved (X axis) versus accumulated time (Y axis)

seconds (only one processor was allowed to run for each run) and the memory limit was set to 500 MB.

The overall results are reported in Fig. 4. The rows show the comparison between MATHSAT and, respectively, CVCLITE, ICS and UCLID, whereas the columns correspond to the different test suites. The X and Y axes show, respectively, MATHSAT and each of the competitor's execution times. A dot in the upper part of a picture, i.e. above the diagonal, means a better performance of MATHSAT and viceversa. The two uppermost horizontal lines and the two rightmost vertical lines represent, respectively, benchmarks that ended in out-of-memory (higher) or timed-out (lower) for, respectively, each of the competitors and MATHSAT.

For the UCLID tests only, the dots on the uppermost horizontal line represent problems on which UCLID could not be run on, because it does not support full LAL; thus, these points are significant only for MATHSAT.

The comparison with CVCLITE shows that MATHSAT performs generally better on the majority of the benchmarks in the SAL suite (CVCLITE timeouts on several of them). On the RTLC suite, the comparison is slightly in favour of CVCLITE for some of the problems whose time ratio is close to 1, however CVCLITE has high computation times and even timeouts a couple of times for a few problems which MATHSAT can solve in less than a second. For the CIRC suite, the comparison is definitely in favour of MATHSAT.

The comparison with ICS shows that ICS is superior on the SAL suite (that is, on its own test suite) on a majority of smaller problems. However, MATHSAT performs better on the most difficult problems in the suite (ICS timeouts on some of them). On the RTLC and CIRC suite MATHSAT performs clearly better than ICS.

Finally, the comparison with UCLID shows a substantial difference of performance in favour of MATHSAT [2].

Fig. 5 shows an overall comparison between MATHSAT and its competitors, where the number of tests solved within the time and memory limits (X axis) is plotted against the accumulated execution time (Y axis). Notice that plots with less samples indicate a higher number of time outs. For this reason, we do not report the data for UCLID, given that it could be run only on a subset of the benchmarks.

On the SAL suite, ICS generally performs the best on most examples; however MATHSAT performs better than ICS on the hardest examples (MATHSAT is able to solve some problems for which ICS timeouts); CVCLITE performance on this suite is definitely worse. On the RTLC suite, the performances of MATHSAT and CVCLITE are quite close to each other, whereas ICS performance is definitely worse. Finally, on the CIRC suite MATHSAT is the the best performer followed by ICS, whereas CVCLITE performs much worse. The last picture shows the results obtained by putting together the data in the three benchmarks suites: overall, MATHSAT and ICS perform clearly better than CVCLITE, with MATHSAT performing better than ICS on the harder problems and ICS performing slightly better on simpler ones.

7 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 it has been devoted a lot of interest in the past. In particular, our approach builds upon and improves our previous work on MATHSAT [5, 4, 7, 6, 14], along the lines described in Sect. 3.

Other related decision procedures are the ones considered in Sect. 6, namely CVCLITE [17, 10], ICS [24, 19] and UCLID [36, 33]. 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 [17]. ICS is a decision procedure for the satisfiability of formulas in a quantifier-free, first-order theory containing both uninterpreted function symbols and interpreted symbols from a set of theories including arithmetic,

[2] UCLID could not be run on some of the problems in SAL and RTLC and on all of the problems in CIRC, because it does not support full LAL, hence the emptiness of the bottom-right picture.

tuples, arrays, and bit-vectors. Finally, UCLID is a tool incorporating a decision procedure for arithmetic of counters, the theories of uninterpreted functions and equality (EUF), separation predicates and arrays. It can also handle limited forms of quantification. In this paper these tools have been compared using the benchmarks falling into the class of 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 [20], a tool using lazy-theorem proving based on SAT-solving, supporting domain-specific procedures for the theories of EUF, linear arithmetic and the theory of arrays, and the tool ZAPATO [9], a tool for counterexample-driven abstraction refinement whose overall architecture is similar to Verifun. The DPLL(T) [21] tool is a decision procedure for the theory of EUF, which, similarly to MATHSAT, is based on a DPLL-like SAT-solver engine coupled with a specialized solver for EUF. A comparison with DPLL(T) in the case of EUF is also planned as future work. ASAP [25], is a decision procedure for quantifier-free Presburger arithmetic (that is, the theory of LAL over non-negative integers) implemented on top of UCLID; a comparison was not possible given that the system is not publicly available. A further relevant system is TSAT++ [35, 3], which is limited, however, to the separation logic fragment of LAL.

Finally, we mention [30], which presents HDPLL, a decision procedure for LAL, specialized to the verification of circuits at RTL level. 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 [30], HDPLL seems to be very competitive, at least for property verification of circuits at RTL level. It would be very interesting to perform a thorough experimental evaluation wrt. MATHSAT (at the moment this was not possible due to unavailability of the tool) and also to investigate the possibility of tuning MATHSAT using some ideas mentioned in the paper.

8 Conclusions and Future Work

In this paper we have presented a new decision procedure for 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. First, the theory solver is incremental and backtrackable, and therefore able to tightly interact with the boolean top level by mimicking its stack-based behaviour; furthermore, it provides explanations in case of conflict, and can carry out deductions that provide truth values to unassigned atoms. Second, we heavily exploit the idea of layering: the satisfiability of theory constraints is evaluated in theories of increasing strength (Equality, Separation logic, Linear Arithmetic over the reals, and Linear Arithmetic over the integers). The idea is to privilege less expensive solvers (for weaker theories), thus reducing the use of more expensive solvers. Finally, static learning and weakened early pruning are also used. We carried out a thorough experimental evaluation of our approach: the 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.

Besides the experiments shown in this paper, we have performed an additional set of experiments to evaluate the impact of the above mentioned improvements on the overall performance of MATHSAT. The results of this evaluation are reported in an extended version of this paper, available at [26].

As future work, we plan to further tune MATHSAT, to investigate the impact of different splitting heuristics taking into account the internal nature of constraints. In addition, we plan to extend MATHSAT to deal with other theories, including non-linear arithmetics, arrays, bitvectors, and a model of memory access. We are investigating an extension of MATHSAT to combinations of theories, in particular EUF and LAL. Finally, we plan to lift SAT-based model checking beyond the boolean case, to the verification of sequential RTL circuits and of hybrid systems.

References

1. W. Ackermann. *Solvable Cases of the Decision Problem*. North Holland Pub. Co., Amsterdam, 1954.
2. A. Armando, C. Castellini, and E. Giunchiglia. SAT-based procedures for temporal reasoning. In *Proc. European Conference on Planning, CP-99*, 1999.
3. A. Armando, C. Castellini, E. Giunchiglia, and M. Maratea. 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.
4. 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.*, volume 2392 of *LNAI*. Springer, July 2002.
5. 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*, volume 2385 of *LNAI*, pages 231–245. Springer, 2002.
6. G. Audemard, M. Bozzano, A. Cimatti, and R. Sebastiani. Verifying Industrial Hybrid Systems with MathSAT. In *Proc. of the 1st CADE-19 Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR'03)*, 2003.
7. G. Audemard, A. Cimatti, A. Korniłowicz, and R. Sebastiani. SAT-Based Bounded Model Checking for Timed Systems. In *Proc. FORTE'02*, volume 2529 of *LNCS*. Springer, 2002.
8. G.J. Badros and A. Borning. The Cassowary linear arithmetic constraint solving algorithm: Interface and implementation. Technical Report UW-CSE-98-06-04, Jun 1998.
9. T. Ball, B. Cook, S.K. Lahiri, and L. Zhang. Zapato: Automatic Theorem Proving for Predicate Abstraction Refinement. In *Proc. CAV'04*, pages 457–461. Springer, 2004.
10. C. Barrett and S. Berezin. CVC Lite: A New Implementation of the Cooperating Validity Checker. In *Proc. CAV'04*, volume 3114 of *LNCS*, pages 515–518. Springer, 2004.
11. R. J. Bayardo, Jr. and R. C. Schrag. Using CSP Look-Back Techniques to Solve Real-World SAT instances. In *Proc. AAAI/IAAI'97*, pages 203–208. AAAI Press, 1997.
12. A. Bockmayr and V. Weispfenning. Solving Numerical Constraints. In *Handbook of Automated Reasoning*, pages 751–842. MIT Press, 2001.
13. A. Borning, K. Marriott, P. Stuckey, and Y. Xiao. Solving linear arithmetic constraints for user interface applications. In *Proc. UIST'97*, pages 87–96. ACM, 1997.
14. M. Bozzano, A. Cimatti, G. Colombini, V. Kirov, and R. Sebastiani. The MathSat solver – a progress report. In *Proc. Workhop on Pragmatics of Decision Procedures in Automated Reasoning 2004 (PDPAR 2004)*, 2004.

15. R. Brinkmann and R. Drechsler. RTL-datapath verification using integer linear programming. In *Proc. ASP-DAC 2002*, pages 741–746. IEEE, 2002.
16. Boris V. Cherkassky and Andrew V. Goldberg. Negative-cycle detection algorithms. *Mathematical Programming*, 85(2):277–311, 1999.
17. CVC, CVCLITE and SVC. `http://verify.stanford.edu/{CVC,CVCL,SVC}`.
18. N. Eén and N. Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing (SAT 2003)*, volume 2919 of *LNCS*, pages 502–518. Springer, 2004.
19. J.-C. Filliâtre, S. Owre, H. Ruess, and N. Shankar. ICS: Integrated Canonizer and Solver. In *Proc. Conference on Computer Aided Verification (CAV'01)*, pages 246–249, 2001.
20. C. Flanagan, R. Joshi, X. Ou, and J.B. Saxe. Theorem Proving using Lazy Proof Explication. In *Proc. CAV'03*, volume 2725 of *LNCS*, pages 355–367. Springer, 2003.
21. H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In *Proc. CAV'04*, volume 3114 of *LNCS*, pages 175–188. Springer, 2004.
22. C. Gomes, B. Selman, and H. Kautz. Boosting Combinatorial Search Through Randomization. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, 1998.
23. I. Horrocks and P. F. Patel-Schneider. FaCT and DLP. In *Proc. Tableaux'98*, pages 27–30, 1998.
24. ICS. `http://www.icansolve.com`.
25. D. Kroening, J. Ouaknine, S. Seshia, and O. Strichman. Abstraction-Based Satisfiability Solving of Presburger Arithmetic. In *Proc. CAV'04*, pages 308–320. Springer, 2004.
26. MATHSAT. `http://mathsat.itc.it`.
27. M. W. Moskewicz, C. F. Madigan, Y. Z., L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference*, 2001.
28. R. Nieuwenhuis and A. Oliveras. Congruence Closure with Integer Offsets. In *Proc. 10th LPAR*, number 2850 in LNAI, pages 77–89. Springer, 2003.
29. Omega. `http://www.cs.umd.edu/projects/omega`.
30. G. Parthasarathy, M.K. Iyer, K.-T. Cheng, and Li-C. Wang. An efficient finite-domain constraint solver for circuits. In *Proc. DAC'04*, pages 212–217. IEEE, 2004.
31. SAL Suite. `http://www.csl.sri.com/users/demoura/gdp-benchmarks.html`.
32. S. Schulz. E – A Brainiac Theorem Prover. *AI Communications*, 15(2/3):111–126, 2002.
33. S.A. Seshia, S.K. Lahiri, and R.E. Bryant. A hybrid SAT-based decision procedure for separation logic with uninterpreted functions. In *Proc. DAC'03*, pages 425–430. ACM, 2003.
34. J. P. M. Silva and K. A. Sakallah. GRASP - A new Search Algorithm for Satisfiability. In *Proc. ICCAD'96*, 1996.
35. TSAT++. `http://www.ai.dist.unige.it/Tsat`.
36. UCLID. `http://www-2.cs.cmu.edu/~uclid`.
37. S. Wolfman and D. Weld. The LPSAT Engine & its Application to Resource Planning. In *Proc. IJCAI*, 1999.
38. L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In *Proc. CAV'02*, volume 2404 of *LNCS*, pages 17–36. Springer, 2002.