A SAT Based Approach for Solving Formulas over Boolean and Linear Mathematical Propositions*

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

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

Abstract. The availability of decision procedures for combinations of boolean and linear mathematical propositions opens the ability to solve problems arising from real-world domains such as verification of timed systems and planning with resources. In this paper we present a general and efficient approach to the problem, based on two main ingredients. The first is a DPLL-based SAT procedure, for dealing efficiently with the propositional component of the problem. The second is a tight integration, within the DPLL architecture, of a set of mathematical deciders for theories of increasing expressive power. A preliminary experimental evaluation shows the potential of the approach.

1 Introduction

The definition of decision procedures for expressive logical theories, in particular theories combining constraints over boolean and real variables, is a very important and challenging problem. Its importance lies in the fact that problems arising from different real-world domains, ranging from formal verification of infinite state systems to planning with resources, can be easily encoded as decision problems for such theories. The challenge is to define automatic decision procedures, that are able to deal with a wide class of problems, but are also able to recognize easy problems and to deal with them efficiently.

In this paper, we tackle the decision problem for boolean combinations of linear mathematical propositions. We propose an approach based on the extension of efficient DPLL decision procedures for propositional satisfiability with a

* This work is sponsored by the CALCULEMUS! IHP-RTN EC project, contract code HPRN-CT-2000-00102, and has thus benefited of the financial contribution of the Commission through the IHP programme. We thank Andrew Goldberg, Stefano Pallottino and Romeo Rizzi for invaluable suggestions about the problems of solving linear (in)equalities.

set of mathematical deciders of increasing power. The approach is general and incremental. It allows for the structured integration of mathematical solvers of different expressive power within the DPLL decision procedure, with constraints learning and backjumping. The mathematical solvers have different expressive power, ranging from equalities, to binary linear inequalities, to full linear inequalities. More complex solvers come into play only when needed.

We implemented the approach in the MATH-SATsolver, based on the SIM package for propositional satisfiability. An experimental evaluation was carried out on tests arising from temporal reasoning [2] and formal verification of timed systems [3]. In the first class of problems, we compare our results with the results of the specialized system; although MATH-SAT is able to tackle a wider class of problems, it runs faster that the TSAT solver, that is specialized to a problem class. In the second class, we show the impact of a tighter degree of integration and the different optimization techniques on the ability of the solver. Although preliminary, the experimental evaluation is extremely promising.

The paper is structured as follows. In Section 2 we formalize the class of problems of interest. In Section 3.1 we discuss the general architecture of the solver, and the specific decision procedures that are currently integrated. In Section 4 we present an experimental evaluation, and in Section 5 we describe some related work and draw some conclusions.

2 MATH-SAT

By *math-terms* and *math-formulas* we denote respectively the linear mathematical expressions and formulas built on constants, variables and arithmetical operators over $\mathbb{R}$ and boolean connectives:

- a constant $c_i \in \mathbb{R}$ is a math-term;
- a variable v_i over $\mathbb{R}$ is a math-term;
- $c_i \cdot v_j$ is a math-term, $c_i \in \mathbb{R}$ and v_j being a constant and a variable over $\mathbb{R}$;
- if t_1 and t_2 are math-terms, then $-t_1$ and $(t_1 \otimes t_2)$ are math-terms, $\otimes \in \{+, -\}$.
- a boolean proposition A_i over $\mathbb{B} := \{\bot, \top\}$ is a math-formula;
- if t_1, t_2 are math-terms, then $(t_1 \bowtie t_2)$ is a math-formula, $\bowtie \in \{=, \neq, >, <, \geq, \leq\}$;
- if φ_1, φ_2 are math-formulas, then $\neg\varphi_1$ and $(\varphi_1 \wedge \varphi_2)$ are math-formulas.

The boolean connectives $\vee$, $\rightarrow$, $\leftrightarrow$ are defined from $\wedge$ and $\neg$ in the standard way. For instance, $A_1 \wedge ((v_1 + 5.0) \leq 2.0 \cdot v_3)$ is a math-formula.

An *atom* is any math-formula in one of the forms A_i or $(t_1 \bowtie t_2)$ above —respectively called *boolean atoms* and *mathematical atoms*. A *literal* is either an atom (a *positive* literal) or its negation (a *negative* literal). If l is a negative literal $\neg\psi$, then by "$\neg l$" we conventionally mean ψ rather than $\neg\neg\psi$. We denote by $Atoms(\phi)$ the set of mathematical atoms of a math-formula ϕ.

By *interpretation* is a map $\mathcal{I}$ which assigns real values and boolean values to math-terms and math-formulas respectively and preserves constants, arithmetical and boolean operators:

- $\mathcal{I}(A_i) \in \{\top, \bot\}$, for every $A_i \in \mathcal{A}$;
- $\mathcal{I}(c_i) = c_i$, for every constant $c_i \in \mathbb{R}$;
- $\mathcal{I}(v_i) \in \mathbb{R}$, for every variable v_i over $\mathbb{R}$;
- $\mathcal{I}(t_1 \otimes t_2) = \mathcal{I}(t_1) \otimes \mathcal{I}(t_2)$, for all math-terms t_1, t_2 and $\otimes \ \in \{+, -, \cdot\}$;
- $\mathcal{I}(t_1 \bowtie t_2) = \mathcal{I}(t_1) \bowtie \mathcal{I}(t_2)$, for all math-terms t_1, t_2 and $\bowtie \in \{=, \neq, >, <, \geq, \leq\}$;
- $\mathcal{I}(\neg\varphi_1) = \neg\mathcal{I}(\varphi_1)$, for every math-formula φ_1;
- $\mathcal{I}(\varphi_1 \wedge \varphi_2) = \mathcal{I}(\varphi_1) \wedge \mathcal{I}(\varphi_2)$, for all math-formulas φ_1, φ_2.

E.g., $\mathcal{I}((v_1 - v_2 \geq 4) \wedge (\neg A_1 \vee (v_1 = v_2)))$ is $(\mathcal{I}(v_1) - \mathcal{I}(v_2) \geq 4) \wedge (\neg\mathcal{I}(A_1) \vee (\mathcal{I}(v_1) = \mathcal{I}(v_2)))$. We say that $\mathcal{I}$ *satisfies* a math formula ϕ, written $\mathcal{I} \models \phi$, iff $\mathcal{I}(\phi)$ evaluates to true. E.g., $A_1 \to ((v_1 + 2v_2) \leq 4.5)$ is satisfied by an interpretation $\mathcal{I}$ s.t. $\mathcal{I}(A_1) = \top$, $\mathcal{I}(v_1) = 1.1$, and $\mathcal{I}(v_2) = 0.6$.

We call *MATH-SAT* the problem of checking the satisfiability of math-formulas. As standard boolean formulas are a strict subcase of math-formulas, it follows trivially that MATH-SAT is NP-hard.

A *truth assignment* for a math-formula ϕ is a truth value assignment μ to (a subset of) the atoms of ϕ. We represent truth assignments as set of literals

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

$\alpha_1, \ldots, \alpha_N, \beta_1, \ldots, \beta_M$ 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 it makes ϕ evaluate to true. We say that an interpretation $\mathcal{I}$ satisfies an assignment μ iff $\mathcal{I}$ satisfies all the elements of μ. For instance, the assignment $\{(v_1 - v_2 \geq 4.0), \neg A_1\}$ propositionally satisfies $(v_1 - v_2 \geq 4.0) \wedge (\neg A_1 \vee (v_1 = v_2))$, and it is satified by $\mathcal{I}$ s.t. $\mathcal{I}(v_1) = 6.0$, $\mathcal{I}(v_2) = 1.0$, $\mathcal{I}(A_1) = \bot$. Intuitively, if we see a math-formula φ as a propositional formulas in its atoms, then $\models_p$ is the standard satisfiability in propositional logic.

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:

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

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

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

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

do not have any satisfying interpretation. ◇

3 The Solver

3.1 General idea

The key idea in our approach to solving the MATH-SAT problem consists in stratifying the problem over N layers $L_0, L_1, \ldots, L_{N-1}$ of increasing complexity, and searching for a solution "at a level as simple as possible". In our view, each level considers only an abstraction of the problem which interprets a subgrammar $G_0, G_1, \ldots, G_{N-1}$ of the original problem, G_{N-1} being the grammar G of the problem. Since L_n refines L_{n-1}, if the problem does not admit a solution at level L_n, then it does not at $L_0, \ldots, L_{n-1}$. If indeed a solution S exists at L_n, either n equals $N-1$, in which case S solves the problem, or a refinement of S must be searched at L_{n+1}. In this way, much of the reasoning can be performed at a high level of abstraction. This results in an increased efficiency in the search of the solution, since low-level searches, which are often responsible for most of the complexity, are avoided whenever possible.

The simple and general idea above maps to an N-layered architecture of the solver. In general, a layer L_n is called by layer L_{n-1} to refine a (maybe partial) solution S of the problem. L_n must check for unsatisfability of S and (a) return failure if no refinement can be found, or (b) invoke L_{n+1} upon a refinement S', unless n equals $N-1$. An explanation for failure can be added in case (a), to help higher levels "not to try the same wrong solution twice". L_0 must behave slightly differently, by enumerating (abstract) solutions.

Our solver MATH-SAT realizes the ideas above over the *MATH-SAT* problem. MATH-SAT works on 5 refinement layers. L_0 takes into account only propositional connectives, and is realized by a DPLL propositional satisfiability procedure, modified to act as an enumerator for propositional assignments. To optimize the search, L_0 does not actually ignore mathematical atoms; rather, it abstracts them into boolean atoms, in order to reason upon them at an abstract level. As such, L_0 incorporates an association between newly introduced boolean atoms and originating mathematical atoms, which is used to communicate with L_1. L_1 considers also equalities, performing equality propagation, building equality-driven clusters of variables and detecting equality-driven unsatisfiabilities. L_2 handles also inequalities of the kind $(v_1 - v_2 \bowtie c)$, $\bowtie \in \{<, >, \leq, \geq\}$, by a variant of the Bellman-Ford minimal path algorithm. L_3 considers also general inequalities —except negated equalities— using a standard simplex algorithm. Finally, L_4 considers also negated equalities.

```
boolean Math-SAT(formula φ, interpretation & I)
    μ = ∅;
    return Math-DPLL(φ, μ, I);

boolean Math-DPLL(formula φ, assignment & μ, interpretation & I)
    if (φ == ⊤) {                                  /* base          */
        I = Math-Solve(μ) ;
        return (I ≠ Null) ; }
    if (φ == ⊥)                                    /* backtrack */
        return False;
    if {a literal l occurs in φ as a unit clause}  /* unit propagation   */
        return Math-DPLL(assign(l, φ), μ ∪ {l}, I);
    l = choose-literal(φ);                         /* split         */
    return   (Math-DPLL(assign(l, φ), μ ∪ {l}, I)   or
              Math-DPLL(assign(¬l, φ), μ ∪ {¬l}, I) );
```

Fig. 1. Pseudo-code of the basic version of the MATH-SAT procedure.

The decomposition in MATH-SAT is significant both because it allows exploiting specialized efficient algorithms to deal with each layer, and because a number of significant problems can be expressed using one of the subgrammars G_0, G_1, G_2. For instance, classical planning problems can be encoded in G_0, both the solving of disjunctive temporal constraints and the reachability of timed systems can be encoded in G_2. In those cases, the specialized search algorithms are used, so efficiency is not sacrificed to expressivity.

3.2 L_0: The boolean solver

To solve the satisfiability problem for our math-formulas, we have implemented a solver based on a variant of DPLL, along the guidelines described in [16]. The basic schema of such a procedure, called MATH-SAT, is reported in Figure 1. MATH-SAT is sound and complete [16].

MATH-SAT takes as input a math-formula φ and returns a truth value asserting whether φ is satisfiable or not, and in the former case an interpretation $\mathcal{I}$ satisfying φ. MATH-SAT is a wrapper for the main routine, MATH-DPLL. MATH-DPLL looks for a truth assignment μ propositionally satisfying φ which is satisfiable from the mathematical viewpoint. This is done recursively, according to the following steps:

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

– (unit) If a literal l occurs in φ as a unit clause, then l must be assigned $\top$.Thus, MATH-DPLL is recursively invoked upon *assign(l, φ)* and the assignment obtained by adding l to μ. *assign(l, φ)* substitutes every occurrence of l in φ with $\top$ and propositionally simplifies the result.
– (split) If none of the above situations occurs, then *choose-literal(φ)* returns an unassigned literal l according to some heuristic criterion. Then MATH-DPLL is first invoked upon *assign(l, φ)* and $\mu \cup \{l\}$. If the result is $False$, then MATH-DPLL is invoked upon *assign($\neg l, \varphi$)* and $\mu \cup \{\neg l\}$.

MATH-DPLL is a variant of DPLL, modified to work as an enumerator of truth assignments, whose satisfiability is recursively checked by MATH-SOLVE. The key difference wrt. standard DPLL is in the “base” step. Standard DPLL needs finding only one satisfying assignment μ, and thus simply returns $True$. MATH-DPLL instead also needs checking the satisfiability of μ, and thus it invokes MATH-SOLVE(μ). Then it returns $True$ if a non-null interpretation satisfying μ is found, it returns $False$ and backtracks otherwise.

The search space of the MATH-SAT problem for a math-formula φ is *infinite*. However, MATH-DPLL partitions such space into a *finite* number of regions, each induced by the mathematical constraints in one assignment μ propositionally satisfying φ. Each such region may contain an up-to-infinite set of satisfying interpretations. If so, MATH-SOLVE picks and returns one of them. Also, since MATH-SOLVE works in polynomial space, MATH-SAT works in polynomial space.

3.3 L_1-L_4: The mathematical solver

MATH-SOLVE takes as input an assignment μ, and returns either an interpretation $\mathcal{I}$ satisfying μ or $Null$ if there is none. (For simplicity we assume to rewrite all the negated mathematical literals in μ into positive atoms, e.g., $\neg(t_1 = t_2) \Longrightarrow (t_1 \neq t_2)$, $\neg(t_1 > t_2) \Longrightarrow (t_1 \leq t_2)$, etc.)

L_1: Eliminating equalities The first step eliminates from μ all equalities and simplifies μ accordingly. First, all atoms in the form $(v_i = v_j)$ are removed from μ and all variables occurring there are collected into equivalence classes $E_1, ..., E_i, ..., E_k$, and for each E_i a representant variable $v'_i \in E_i$ is designated. Then, for each E_i, all variables in E_i are substituted by their representant v'_i in the mathematical atoms of μ. All valid atoms (like, e.g., $(v_i - v_i \neq 2), (v_i - v_i > -1)$) are removed, together with all duplicated atoms. If an inconsistent atom is found (like, e.g., $(v_i - v_i = 2), (v_i - v_i \leq -1)$) then MATH-SOLVE terminates returning $Null$.

Second, each remaining atom in the form $(v_i = ...)$ in μ is removed, by applying equality propagation. Throughout this phase, all valid and duplicated atoms are removed, and, if an inconsistent atom is found, then MATH-SOLVE terminates returning $Null$.

L_2: Minimal path plus negative cycle detection If only atoms in the form $(v_i - v_j \bowtie c)$ are left, $\bowtie \in \{>, <, \geq, \leq\}$, then the resulting problem is solved by invoking a minimum path algorithm with cycle detection, a variant of the Bellman-Ford algorithm described in [8], which either returns a satisfying interpretation for the variables v_i's or verifies there is none. In the former case MATH-SOLVE decodes back the resulting interpretation and returns it, otherwise it returns $Null$. The algorithm is worst-case quadratic in time and linear in size.

L_3: Linear programming Otherwise —unless some negated equality $(t_i \neq t_j)$ exist — a linear programming (LP) simplex algorithm is invoked, which, again, either returns a satisfying interpretation for the variables v_i's or verifies there is none. MATH-SOLVE behaves as in the previous case. This algorithm is worst-case exponential in time (but it is well-known that it exhibits polynomial behavior in non-pathological practical cases) and always requires polynomial memory.

L_4: Handling negated equalities Neither minimal path nor LP procedures handle negated equality constraints like $(t_i \neq t_j)$. In many significant cases — including the ones of practical interest for us, see Section 4— it is always possible to avoid them.

However, in order to preserve expressiveness our design handles them. A trivial way to handle the problem is to split every negated equalities $(t_i \neq t_j)$ into the disjunction of the corresponding strict inequalities $(t_i > t_j) \vee (t_i < t_j)$, and handle the distinct problems separately. This is, of course, rather inefficient.

Instead, we first ignore negated equalities and run one of the algorithms on the remaining part. (i) if there is no solution, then the problem is unsatisfiable anyway, so that it is returned $Null$; (ii) if a solution $\mathcal{I}$ is found, then it is checked against the negated equalities: if it does not contradict them, then $\mathcal{I}$ is returned; (iii) if not, the negated equalities are split and the resulting problems are analyzed.

Notice that the latter event is extremely rare, for two reasons. First, MATH-SOLVE finds a solution at most once in the whole computation. All the other problems are unsatisfiable. Second, a constraint like $(t_i \neq t_j)$ covers only a null-measuring portion of the space of the variables in t_1 and t_2, while the space of the solutions of a solvable linear problem is typically a polyhedron containing infinitely many solutions. Thus $(t_i \neq t_j)$ makes a solvable problem unsolvable only if the solution space degenerates into a n-dimensional point.

3.4 Improvements & optimizations

We describe some improvements and optimizations for MATH-SAT, some of which come from adapting to our domain improvements and optimizations of the DPLL-based procedures for modal logics [12, 14, 13] and for temporal reasoning and resource planning [2, 19].

Preprocessing atoms One potential source of inefficiency for the procedure of Figure 1 is the fact that semantically equivalent but syntactically different atoms are not recognized to be identical [resp. one the negation of the other] and thus they may be assigned different [resp. identical] truth values. This causes the undesired generation of a potentially very big amount of intrinsically unsatisfiable assignments (for instance, up to $2^{Atoms(\varphi)-2}$ assignments of the kind $\{(v_1 < v_2), (v_1 \geq v_2), ...\}$).

To avoid these problems, it is wise to preprocess atoms so that to map semantically equivalent atoms into syntactically identical ones:

- *exploit associativity* (e.g., $(v_1 + (v_2 + v_3))$ and $((v_1 + v_2) + v_3)) \Longrightarrow (v_1 + v_2 + v_3)$);
- *sorting* (e.g., $(v_1 + v_2 \leq v_3 + 1)$, $(v_2 + v_1 - 1 \leq v_3) \Longrightarrow (v_1 + v_2 - v_3 \leq 1)$);
- *exploiting negation* (e.g., $(v_1 < v_2)$, $(v_1 \geq v_2) \Longrightarrow (v_1 < v_2)$, $\neg(v_1 < v_2)$).

Early Pruning (EP) If an assignment μ' is unsatisfiable, then all its supersets are unsatisfiable. If the unsatisfiability of an assignment μ' is detected during its recursive construction, then this prevents checking the satisfiability of all the up to $2^{|Atoms(\varphi)|-|\mu'|}$ truth assignments which extend μ'.

This suggests to introduce an intermediate satisfiability test on incomplete assignments just before the "split" step:

if *Likely-Unsatisfiable(μ)* /* early pruning */
    if (MATH-SOLVE(μ) = *Null*)
        then return *False;*

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

In this case MATH-SOLVE needs not returning explicitly the interpretation $\mathcal{I}$, so that it can avoid decoding back the solution found by the solver. Moreover, negated equalities $(t_i \neq t_j)$, if any, can be ignored here, as they may only make a satisfiable problem unsatisfiable, not vice versa.

Example 2. Consider the formula φ of Example 1. Suppose that, in four recursive calls, MATH-DPLL builds, in order, the intermediate assignment:

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

(rows 1, 2, 3 and 4 of φ), which contains the conflict set (2) and is thus unsatisfiable. If MATH-SOLVE is invoked on μ', it returns $Null$, and MATH-DPLL backtracks without exploring any extension of μ'. $\diamond$

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

Enhanced early pruning (EEP) In early pruning, the call to MATH-SOLVE is not effective if μ is satisfiable. Anyway such a call can produce information which can be used to reduce search afterwords. In fact, the mathematical analysis of

μ performed by MATH-SOLVE can allow to assign deterministically truth values to some mathematical atoms $\psi \notin \mu$, and this information can be returned by MATH-SOLVE as a new assignment η, which is unit-propagated away by MATH-DPLL.

For instance, assume that all the following mathematical atoms occur in the math-formula. If $(v_1 - v_2 \leq 4) \in \mu$ and $(v_1 - v_2 \leq 6) \notin \mu$, then MATH-SOLVE can derive deterministically that the latter is true, and thus return an assignment η containing $(v_1 - v_2 \leq 6)$. Similarly, if $(v_1 - v_2 > 2) \notin \mu$ and MATH-SOLVE(μ) finds that v_1 and v_2 belong to the same equivalence class, then it η contains $\neg(v_1 - v_2 > 2)$.

(Mathematical) Backjumping (BJ) An alternative optimization starts from the same observations as those of early pruning. Any branch containing a conflict set is unsatisfiable. Thus, suppose MATH-SOLVE is modified to return also a conflict set η causing the unsatisfiability of the input assignment μ. (As for L_2, a negative cycle represents a conflict set; as for L_3, a technique for returning a conflict sets in LP is hinted in [19].) If so, MATH-DPLL can jump back in its search to the deepest branching point in which a literal $l \in \eta$ is assigned a truth value, pruning the search space below.

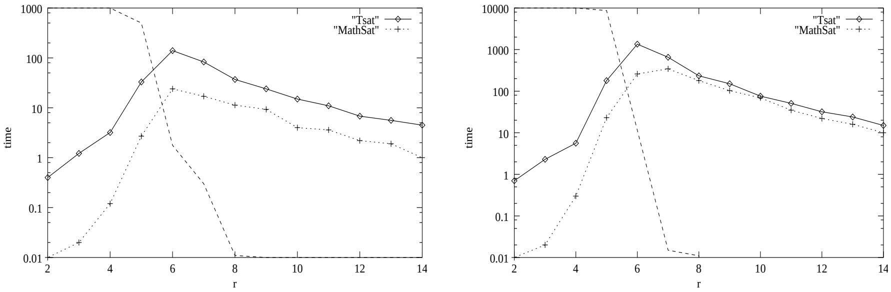
Notice the difference w.r.t. early pruning. Both prune the search tree under a branch containing a conflict set. On one hand, backjumping invokes MATH-SOLVE only at the end of the branch, avoiding useless calls. On the other hand, early pruning prunes the search as soon as there is one conflict set in the assignment, whilst backjumping can prune a smaller search tree, as the conflict set returned by MATH-SAT is not necessarily the one which causes the highest backtracking.

Example 3. Consider the formula φ and the assignment μ of Example 1. Suppose that MATH-DPLL generates μ following the order of occurrence within φ, and that MATH-SOLVE(μ) returns the conflict set (2). Thus MATH-DPLL can backjump directly to the branching point $\neg(3v_1 - v_3 \leq 6)$ without branching first on $(v_3 = 3v_5 + 4)$ and $\neg(2v_2 - v_3 > 2)$, obtaining the same pruning effect as in example 2. If instead MATH-SOLVE(μ) returns the conflict set (3), forcing a branch on $(v_3 = 3v_5 + 4)$. $\diamond$

(Mathematical) Learning When MATH-SOLVE returns a conflict set η, the clause $\neg\eta$ can be added in conjunction to φ. Since then, MATH-DPLL will never again generate any branch containing η.

Example 4. As in Example 3, suppose MATH-SOLVE(μ) returns the conflict set (2). Then the clause $\neg(3v_1 - 2v_2 \leq 3) \vee (2v_2 - v_3 > 2) \vee (3v_1 - v_3 \leq 6)$ is added in conjunction to φ. Thus, whenever a branch contains two elements of (2), then MATH-DPLL will assign the third to $\bot$ by unit propagation. $\diamond$

Learning is a technique which must be used with some care, as it may cause an explosion in size of φ. To avoid this, one has to introduce techniques for discarding learned clauses when necessary [4].



25 vars 30 vars

Fig. 2. Comparison between TSAT and MATH-SAT. $k = 2$, $n = 25, 30$, $L = 100$, $r := m/n$ in $[2, ..., 14]$. 100 sample formulas per point. Median CPU times (secs). Background: satisfiability rate.

Notice the difference w.r.t. standard boolean backjumping and learning [4]. In the latter case, the conflict set propositionally falsifies the formula, while in our case it is inconsistent from the mathematical viewpoint.

Triggering This technique is a generalization we propose of a technique adopted in [19]. It comes from the consideration (proved in [16]) that, if we have mathematical atoms occurring only positively [resp. negatively] in the input formulas, we can drop any negative [positive] occurrence of them from the assignment to be checked by MATH-SOLVE. This is particularly useful when we deal with equality atoms occurring only positively, as it avoids handling negated equalities.

4 Some experimental results

We've implemented MATH-SAT in C; MATH-DPLL is built on top of the SIM library [11]; MATH-SOLVE uses alternatively a home-made implementation of Bellman-Ford minimal path algorithm with negative cycle detection [8], and the Simplex LP library LP_SOLVE [5], as described in Section 3.3.

All experiments presented here were run under `Linux RedHat 7.1` on a 4-processor `PentiumIII` 700MHz machine with more than 4GB RAM, with a time limit of 1 hour and a RAM limit of 1GB for each run. (All the math-formulas investigated here are available at `http://www.science.unitn.it/~rseba/Mathsat.html`, together with our implementation of MATH-SAT.)

4.1 Temporal Reasoning

As a first application example, we consider one of the most studied problems in the domain of temporal reasoning, that of solving the consistency of *disjunctive temporal problems* (DTP). Following [18], we encode the problem as a particular a MATH-SAT problem, where the math-formulas are in the restricted form:

$$\bigwedge_i \bigvee_j (v_{1_{ij}} - v_{2_{ij}} \leq c_{ij}), \tag{5}$$

$v_{k_{ij}}$ and c_{ij} being real variables and integer constants respectively. Notice that here (i) there are no boolean variables (ii) constraints are always in the form $(v_i - v_j \leq c)$ and (iii) they always occur positively.

[18] proposed as a benchmark a random generation model in which DTPs are generated in terms of four integer parameters k, m, n, L: a DTP is produced by randomly generating m distinct clauses of length k of the form (5); each atom is obtained by picking $v_{1_{ij}}$ and $v_{2_{ij}}$ with uniform probability $1/n$ and $c_{ij} \in [-L, L]$ with uniform probability $1/(2L+1)$. Atoms containing the same variable like $(v_i - v_i \leq c)$ and clauses containing identical disjuncts are discharged.

[2] presented TSAT, a SAT based procedure ad hoc for DTPs like (5) based on Bohm SAT procedure [7] and the Simplex LP library LP_SOLVE [5]. In the empirical testing conducted on the same benchmarks as in [18], TSAT outperformed the procedure of [18]. TSAT is enhanced by a form of forward checking and of static learning, in which it learns binary constraints corresponding to pairs of mutually-inconsistent mathematical atoms.

We have run TSAT and MATH-SAT with enhanced early pruning and mathematical learning on the two hardest problems in [2]. The results are reported in Figure 2. As with TSAT, MATH-SAT curves have a peak around the value of r in which we have a 50% satisfiability rate. MATH-SAT is always faster than TSAT, up to one order of magnitude. Similarly to what happens with the optimizations of TSAT [2], when dropping either enhanced early pruning or mathematical learning in MATH-SAT the performances worsen significantly. Thus, although MATH-SAT is a general-purpose procedure, it turns out to be competitive —and even faster— than a current state-of-the-art specific procedure for this problem.

4.2 Model Checking properties of timed systems

As a second application example, we consider the verification and debugging of properties for timed timed systems (e.g., real-time protocols). In short, a timed system is represented as a timed automaton [1], that is, an automaton augmented by real clock variables x, clock resetting statements in the form $(x := 0)$ and clock constraints in the form $(x \bowtie c)$, $\bowtie \in \{>, <, \geq, \leq\}$. The automaton can perform either instantaneous transitions, which are conditioned by clock constraint and can affect the value of boolean variables and resetting statements, or time elapse transition, which increment all clocks by the same value δ and keeps all values.

In [3] we have extended to timed systems the notion of bounded model checking (BMC) [6] and presented a way to encode such problem into a MATH-SAT problem. Given an automaton A, an LTL property f and an integer bound k, we consider the problem of finding an execution of A of up to length k verifying f, and we encode it into the satisfiability of a CNF math-formula $[[A, f]]_k$, s.t. any interpretation of $[[A, f]]_k$ corresponds to a desired execution path.

We introduce a new real variable z representing the current value of $zero$, and we rewrite every occurrence of a clock x with the difference $x - z$. Then, we replicate the propositional and real variables from 0 to k —e.g., x_i represents the variable x at step i— and "unroll" the transition relation from step 0 to step

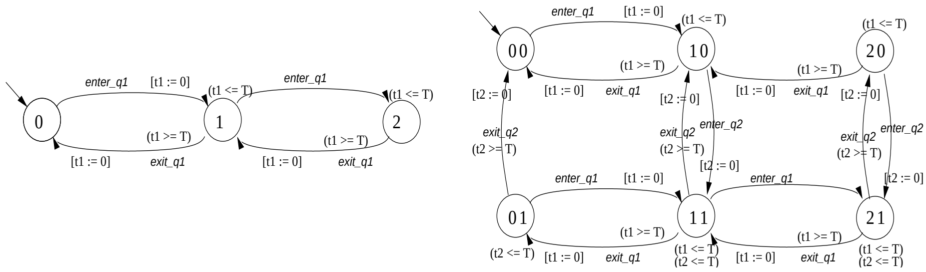


Fig. 3. Timed automata for the post-office problem, for N=1 (left) and N=2 (right).

$k-1$, so that we have

$$[[A, f]]_k := I_0 \wedge \bigwedge_{i=0}^{k-1} T_{i,i+1} \wedge \ [[f]]_k. \tag{6}$$

I_0 is a math formula over the variables at step 0 representing the initial states; $T_{i,i+1}$ is a math formula over the variables at steps i and $i+1$ representing the transition relation; $[[f]]_k$ is a math formula over all the variables from step 0 to step k representing the condition that the path must verify the LTL formula f. (See [6, 3, 9] for details.) The mathematical atoms in $[[A, f]]_k$ are all of the kind $(x = y)$, $(x_i - z_i = x_j - z_j)$ or $(x - y \bowtie c)$, $\bowtie \in \{\leq, \geq, <, >\}$, such that:

1. every atom in the form $(x_i - z_i = x_j - z_j)$ occurs only positively in $[[A, f]]_k$,
2. for every atom $(x = y)$ in $[[A, f]]_k$, there is a corresponding atom $(x > y)$ in $[[A, f]]_k$ s.t. $\neg(x = y) \rightarrow (x > y)$ is a clause in $[[A, f]]_k$.

We have customized a version of MATH-SAT explicitly for this kind of problems. First, MATH-SOLVE(μ) ignores every negated equality in μ. In fact, by point 1., literals like $\neg(x_i - z_i = x_j - z_j)$ can be ignored because of triggering, and, by point 2., literals like $\neg(x = y)$ can be ignored because they are subsumed by $(x > y)$. Second, following [10, 17], the encoder provides some semantic information on the boolean variable, so that the heuristic function *choose_literal()* of Figure 1 can split first on variables labeling transitions, and in order of their step index.

In [3] we presented some empirical tests on timed protocol verification, in which the approach of encoding such problems into math-formulas and running MATH-SAT over them turned out to be competitive wrt. using state-of-the-art verification systems. There the focus was on the encoding, and the goal was to show the effectiveness of our approach w.r.t. other verification approaches. Here instead we focus specifically on the effectiveness of MATH-SAT in solving math-formulas. As we are not aware of any existing decision procedure able to handle efficiently math-formulas of this kind, we restrict to showing how the various variants and optimizations affect the efficiency of MATH-SAT on this kind of formulas on an example.

$k\backslash N$	Basic				Basic+BJ				Basic+EP				Basic+EEP			
	2	3	4	5	2	3	4	5	2	3	4	5	2	3	4	5
2	0.01	0.05	0.24	1.29	0.01	0.05	0.25	1.29	0.01	0.04	0.23	1.31	0.10	0.05	0.25	1.28
3	0.02	0.08	0.37	2.01	0.01	0.08	0.35	2.01	0.02	0.07	0.36	1.91	0.01	0.08	0.38	2.01
4	0.03	0.10	0.51	2.92	0.02	0.09	0.50	2.95	0.02	0.10	0.51	2.92	0.02	0.10	0.51	2.83
5	0.03	0.13	0.62	5.07	0.03	0.13	0.68	5.06	0.03	0.13	0.65	5.00	0.03	0.14	0.68	5.08
6	0.04	0.17	0.90	8.96	0.03	0.16	0.90	8.80	0.03	0.17	0.88	8.80	0.04	0.18	0.89	8.90
7	0.04	0.24	1.82	37	0.05	0.24	1.80	35	0.05	0.23	1.78	37	0.05	0.25	1.83	36
8	0.05	0.41	4.59	231	0.05	0.41	4.54	230	0.06	0.41	4.53	232	0.06	0.40	4.64	229
9		0.92	16	950		0.90	15	945		0.80	14	913		0.86	15	916
10		0.99	72	≥1h		0.98	73	≥1h		0.85	68	≥1h		0.83	70	≥1h
11			302	≥1h			299	≥1h			276	≥1h			288	≥1h
12			592	≥1h			592	≥1h			502	≥1h			529	≥1h
13				≥1h				≥1h				≥1h				≥1h
14				≥1h				≥1h				≥1h				≥1h
Σ	0.22	3.09	991	≥5h	0.20	3.04	990	≥5h	0.22	2.80	870	≥5h	0.22	2.89	911	≥5h

Table 1. MATH-SAT CPU times for the Post-office problem with various optimizations

$k\backslash N$	Basic				Basic+BJ				Basic+EP				Basic+EEP			
	2	3	4	5	2	3	4	5	2	3	4	5	2	3	4	5
2	0.01	0.05	0.25	1.28	0.01	0.05	0.24	1.29	0.01	0.05	0.24	1.30	0.01	0.05	0.24	1.30
3	0.01	0.08	0.38	1.98	0.02	0.07	0.37	1.98	0.01	0.07	0.36	2.05	0.01	0.07	0.37	1.97
4	0.02	0.10	0.49	2.65	0.02	0.10	0.49	2.65	0.02	0.10	0.50	2.73	0.02	0.10	0.49	2.68
5	0.02	0.13	0.63	3.48	0.03	0.13	0.64	3.45	0.03	0.13	0.62	3.51	0.03	0.13	0.61	3.48
6	0.03	0.16	0.74	4.27	0.03	0.16	0.75	4.22	0.03	0.16	0.76	4.33	0.03	0.15	0.78	4.26
7	0.04	0.19	0.88	5.21	0.05	0.19	0.91	5.13	0.04	0.19	0.87	5.19	0.04	0.19	0.90	5.02
8	0.04	0.24	1.09	6.46	0.03	0.23	1.10	6.43	0.05	0.22	1.07	6.40	0.05	0.23	1.07	6.10
9		0.36	1.48	8.98		0.36	1.44	9.08		0.28	1.28	8.15		0.28	1.29	7.54
10		0.29	2.78	16		0.29	2.81	16		0.28	1.83	11		0.28	1.80	9.97
11			8.43	42			8.45	42			2.89	19			2.94	15
12			5.10	159			5.03	155			2.07	47			1.80	33
13				685				742				115				80
14				208				207				38				23
Σ	0.17	1.60	22.2	1145	0.2	1.58	22.2	1199	0.19	1.48	12.5	265	0.19	1.48	12.3	194

Table 2. MATH-SAT CPU times for the Post-office problem with customized *choose_literal* and various optimizations

Consider a post office with N desks, each desk serving a customer every T seconds. Every new customer chooses the desk with shorter queue and, when more than one queue has minimal length, the minimal queue with the minimum index. It is not possible to change a queue after entering it. We want to prove that, although all desks have the same serving time T and customers are "smart", one customer can get into the annoying situation of finding himself in queue after one person, whilst all other queues are empty, and having to wait for a non-instantaneous period in this situation.

The corresponding timing automata for $N = 1$ and $N = 2$ are represented in Figure 3. Each location is labeled by N integers $l_1 l_2 ... l_N$, representing respectively the lengths of queues 1, 2, ... , N. For each queue i a clock variable t_i counts the serving time of the currently served customer. The property is encoded as

$$(20...0)_{k-1} \wedge (20...0)_k \wedge (\delta_{k-1} > 0)$$

for some step $k > 1$, that is, the system is in location 20...0 at both steps $k-1$ and k and a non-null amount of time δ_{k-1} elapses between these two steps. We fix to 2 the maximum queue length of queue 1 and to 1 for the others; this will be enough to show that the problem has a solution, provided the queuing policy of customers. Although the problem is very simple, the size of the automata grow as $3 \cdot 2^N$.

We encoded this problem as described in [3], for increasing values of k and N, and we ran different versions of MATH-SAT on the resulting formulas. The resulting CPU times are reported in Tables 1 and 2. The last row Σ of each table represents the sum of the values in the corresponding column. Table 2 differs from Table 1 for the fact that in the latter we have used the customized version of *choose_literal()* described above.

All problems are unsatisfiable for $k < 2N+4$ and satisfiable for $k \geq 2N+4$, as the minimum path satisfying the property has length $2N+4$. If we consider the case $N=2$ of Figure 3, such a path is:

$$00 \stackrel{enter_{q1}}{\longrightarrow} 10 \stackrel{enter_{q2}}{\longrightarrow} 11 \stackrel{enter_{q1}}{\longrightarrow} 21 \stackrel{\delta=T}{\longrightarrow} 21 \stackrel{served_{q1}}{\longrightarrow} 11 \stackrel{enter_{q1}}{\longrightarrow} 21 \stackrel{served_{q2}}{\longrightarrow} 20 \stackrel{\delta=T}{\longrightarrow} 20$$

that is, customers C1,C2,C3 enter at time 0; after T seconds three new events occur sequentially within a null amount of time: C1 is served by desk 1, a new customer C4 enters queue 1 (as both queues are of length 1), and C2 is served by desk 2. After this, customers C3 and C4 are in queue 1 while nobody in queuing at desk 2, and C4 will have to wait another T seconds before starting being served.

The CPU times in Tables 1 and 2 suggest the following considerations.

First, MATH-SAT with the customized heuristic *choose_literal()* —which chooses transition variables first in order of their step index— dramatically outperforms the basic version, no matter the other optimizations. In fact, as in [10], initial states and transitions are the only sources of non-determinism: once their values are set, the truth values of all other atoms are either irrelevant or derive deterministically from them. This may drastically restrict the search space. Moreover, as in [17], selecting the transitions in forward [backward] step order allows to avoid selecting transition in intermediate steps whose firing conditions are not reachable from the initial states [resp. from whose consequences the goal states are not reachable].

Second, math backjumping is ineffective on these tests. This is due to the fact that, unlike with DTPs, with these math-formulas the conflict sets returned by MATH-SOLVE are very long and nearly always useless for both math backjumping and learning. In fact, when a (possibly small) conflict set is returned by Bellman-Ford, there is no obvious way to reconstruct back the corresponding minimum conflict set by undoing the variable substitutions over the equivalence classes.

Third, simple and enhanced early pruning improve CPU times slightly in Table 1 and very relevantly in Table 2. In particular, this synergy with the customized heuristics *choose_literal()* is due to the fact that the latter can often choose a "bad" transition whose mathematical prerequisites are not verified in the current state. Without early pruning, this causes the generation of a whole search tree of mathematically inconsistent assignments, whose inconsistency is verified one by one. With early pruning, MATH-DPLL invokes MATH-SOLVE and backtracks just after one sequence of unit propagations.

Fourth, whilst enhanced early pruning seems not faster that simple early pruning in the results of Table 1, a significant improvements appears in Table 2. This synergy with the customized heuristics *choose_literal()* is due to the fact that in many situations the value of a clock x is zero —$(x = z)$ in our encoding— because of either resetting or propagating a previous zero value. If so, performing an enhanced early pruning test before choosing the next transition allows to falsify all incompatible transition prerequisites on x —like, e.g., $(x - z \geq T)$— and thus to avoid choosing the corresponding "bad" transition.

A final consideration on the effectiveness of our layered architecture arises as soon as we do some profiling: a significant number of MATH-SOLVE calls —about 70% with basic MATH-SAT, 20-30% with EP, about 10% with EEP— are solved directly by propagating equalities (L_1), without calling Bellman-Ford (L_2).

5 Related work and Conclusions

In this paper we have presented a new approach to the solution of decision problems for combinations of boolean propositions and linear equalities and inequalities over real variables. The approach is general, since it allows to integrate different levels of mathematical solving. This also allows for a significant degree of efficiency, since the more expensive solvers are called only when needed by the subproblem being analyzed. For instance, MATH-SAT is faster than TSAT on the specific class of problems for which the latter has been developed. The other closest related work is [19], where the LPSAT solver is presented, handling math-formulas in which all mathematical atoms occur only positively. The approach, however, is hardwired to the domain of planning, and there is no reference to the architectural issues. In [15], a data structure based on Binary Decision Diagrams (BDDs), combining boolean and mathematical constraints, is used to represent the state space of timed automata. The approach is sometimes very efficient, but it inherits the worst-case exponential memory requirements from BDDs.

In the future, we plan to extend the work presented in this paper along the following directions. First, we will tighten the integration of the different solvers within the SAT architecture. This will allow to incrementally construct the equivalence classes for equality reasoning, and reuse the previously constructed information. Then, we will explore the extensions of the approach to more complex (i.e., quadratic) mathematical constraints, and their applications to formal verification of programs.

References

1. R. Alur. Timed Automata. In *Proc. 11th International Computer Aided Verification Conference*, pages 8–22, 1999.
2. A. Armando, C. Castellini, and E. Giunchiglia. SAT-based procedures for temporal reasoning. In *Proc. European Conference on Planning, ECP-99*, 1999.
3. G. Audemard, A. Cimatti, A. Korniłowicz, and R. Sebastiani. Bounded Model Checking for Timed Systems. Technical Report 0201-05, ITC-IRST, Trento, Italy, January 2002. Submitted for publication.
4. R. J. Bayardo, Jr. and R. C. Schrag. Using CSP Look-Back Techniques to Solve Real-World SAT instances. In *Proc AAAI'97*, pages 203–208. AAAI Press, 1997.
5. Michel Berkelaar. The solver lp_solve for Linear Programming and Mixed-Integer Problems. Available at `http://elib.zib.de/pub/Packages/mathprog/linprog/lp-solve/`.
6. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. CAV'99*, 1999.
7. M. Buro and H. Buning. Report on a SAT competition. Technical Report 110, University of Paderborn, Germany, November 1992.
8. Boris V. Cherkassky and Andrew V. Goldberg. Negative-cycle detection algorithms. *Mathematical Programming*, 85(2):277–311, 1999.
9. A. Cimatti, M. Pistore, M. Roveri, and R. Sebastiani. Improving the Encoding of LTL Model Checking into SAT. In *Proc. 3rd International Workshop on Verification, Model Checking, and Abstract Interpretation*, volume 2294 of *LNCS*. Springer, 2002.
10. E. Giunchiglia, A. Massarotto, and R. Sebastiani. Act, and the Rest Will Follow: Exploiting Determinism in Planning as Satisfiability. In *Proc. AAAI'98*, pages 948–953, 1998.
11. E. Giunchiglia, M. Narizzano, A. Tacchella, and M. Vardi. Towards an Efficient Library for SAT: a Manifesto. In *Proc. SAT 2001*, Electronics Notes in Discrete Mathematics. Elsevier Science., 2001.
12. F. Giunchiglia and R. Sebastiani. Building decision procedures for modal logics from propositional decision procedures - the case study of modal K. In *Proc. CADE13*, LNAI. Springer Verlag, August 1996.
13. F. Giunchiglia and R. Sebastiani. Building decision procedures for modal logics from propositional decision procedures - the case study of modal K(m). *Information and Computation*, 162(1/2), October/November 2000.
14. I. Horrocks and P. F. Patel-Schneider. FaCT and DLP. In *Proc. of Tableaux'98*, number 1397 in LNAI, pages 27–30. Springer-Verlag, 1998.
15. J. Moeller, J. Lichtenberg, H. Andersen, and H. Hulgaard. Fully Symbolic Model Checking of Timed Systems using Difference Decision Diagrams. In *Electronic Notes in Theoretical Computer Science*, volume 23. Elsevier Science, 2001.
16. R. Sebastiani. Integrating SAT Solvers with Math Reasoners: Foundations and Basic Algorithms. Technical Report 0111-22, ITC-IRST, November 2001.
17. Ofer Shtrichmann. Tuning SAT Checkers for Bounded Model Checking. In *Proc. CAV'2000*, volume 1855 of *LNCS*. Springer, 2000.
18. K. Stergiou and M. Koubarakis. Backtracking algorithms for disjunctions of temporal constraints. In *Proc. AAAI*, pages 248–253, 1998.
19. S. Wolfman and D. Weld. The LPSAT Engine & its Application to Resource Planning. In *Proc. IJCAI*, 1999.