The MathSAT Solver — a progress report

(Extended Abstract)

Marco Bozzano[1], Alessandro Cimatti[1], Gabriele Colombini[2],
Veselin Kirov[1], Roberto Sebastiani[1,2]
[1] ITC-IRST, via Sommarive 16, 38050 Povo, Trento, Italy
{bozzano,cimatti,kirov}@itc.it,
[2] Università di Trento, via Sommarive 14, 38050 Povo, Trento, Italy
colombin@kirk.science.unitn.it, roberto.sebastiani@dit.unitn.it

May 28, 2004

Abstract

Many problems of practical relevance are conveniently expressed as boolean combinations of propositional variables and mathematical constraints. The development of decision procedures able to check the satisfiability of such formulas is therefore being devoted an increasing interest.

The MATHSAT family of deciders is based on the extension of a DPLL propositional satisfiability procedure, used as an assignment enumerator. MATHSAT pioneers a lazy and layered approach, where propositional reasoning is tightly integrated with solvers of increasing expressive power (e.g. to reason about equality and linear arithmetic) in such a way that "more expensive" layers are called less frequently.

In this paper, we show the advances in the development of MATHSAT. We discuss the implications related to the use of MINISAT, a new-generation propositional SAT solver; the role of an incremental mathematical reasoner; the role of static learning; and the extension to integer variables. We show that the new version of MATHSAT is significantly more efficient than the previous one.

1 Introduction

Many problems of practical relevance (e.g. equivalence checking and model checking of RTL designs, verification of timed and hybrid systems, temporal planning) are not easily expressible in propositional logic. Rather, they can be conveniently expressed in extensions of propositional logics, with boolean combinations of propositional variables and mathematical constraints. For instance, the timing constraints typical of timed automata can be expressed with inequalities over differences of real variables.

The development of decision procedures able to check the satisfiability of such extended formulas is therefore being devoted an increasing interest, along two parallel but related lines of activity. The *eager* approach (see for instance [SLB03] and [SSB02]) is based on encoding the satisfiability of an extended theory to a propositional satisfiability problem, with the introduction of new axioms that encode the required properties into the boolean logic. The approach is called eager since all the theory is lifted to the boolean level before the search is started. The crucial problem of the eager approach is indeed the containment of the propositional theory

resulting from the encoding; however, once the encoding is produced, solving can be carried out by an efficient, off-the-shelf propositional solver. In the *lazy* approach [ACG99, BDS02, dMRS02, FJOS03], on the other hand, theory reasoning is invoked on demand, e.g. to decide the consistency of mathematical theories resulting from assignment of truth values. This may have an advantage in the cases where the eager approach fails to generate encodings of reasonable size; however, solvers are conceptually more complex than purely propositional engines, and it is crucial to optimize the interplay between the boolean component and the mathematical reasoners.

MATHSAT [ABC+02] is a family of lazy solvers, based on the integration of theory deciders into a DPLL propositional satisfiability procedure, which is used as an assignment enumerator. MATHSAT pioneers a *layered* approach, where propositional reasoning is tightly integrated with solvers of increasing expressive power (e.g. to reason about equality and linear arithmetic) in such a way that "more expensive" layers are called less frequently. Furthermore, compared to other approaches, MATHSAT leverages counterexamples from inconsistencies detected during mathematical reasoning. MATHSAT has been applied in bounded model checking of timed [ACKS02] and hybrid systems [ABCS03], and is being used for the verification of RTL hardware designs.

In this paper, we present the recent advances in the development of MATHSAT. In particular, we discuss the implications related to the use of MINISAT [ES03], a new-generation propositional SAT solver, with respect to the previous pre-Chaff solver SIM [GMTZ01]; the use of static learning; and the role of an incremental mathematical reasoner. We show that the new version of MATHSAT is significantly more efficient than the previous one.

The paper is organized as follows. In Section 2 we give some background on the MATHSAT approach and solver. In Section 3 we describe the advances discussed in this paper. In Section 4 we present a preliminary experimental evaluation (a more accurate analysis will be available in the final version of this paper). In Section 5 we draw some conclusions, and outline the ongoing work and the future directions.

2 MATHSAT version 1: the starting point

We call a *math-formula* a boolean combination of boolean variables and linear constraints over numerical variables. We call an *interpretation* a map $\mathcal{I}$ which assigns real and boolean values to real and boolean variables respectively and preserves constant values, arithmetical and boolean operators. We call MATHSAT the problem of checking the satisfiability of a math-formula. We call a *truth assignment* for a math-formula ϕ a truth value assignment μ to (a subset of) the atoms of ϕ. We say that μ *propositionally satisfies* ϕ, written $\mu \models_p \phi$, iff it makes ϕ evaluate to true. We represent truth assignments as sets of literals, with the intended meaning that positive and negative literals represent atoms assigned to true and to false respectively. $\mathcal{I}$ satisfies μ iff it satisfies all its elements.

To solve the MATHSAT problem, we have implemented MATHSAT [ABC+02], a solver based on a variant of the DPLL SAT procedure [DLL62]. The basic schema of such a procedure is reported in Figure 1. MATHSAT takes as input a math-formula φ, expressed in CNF, and (by reference) an empty interpretation $\mathcal{I}$, and returns a truth value asserting whether φ is satisfiable or not, $\mathcal{I}$ being respectively an interpretation satisfying φ or $Null$. MATHSAT invokes MATH-DPLL passing as arguments φ and (by reference) an empty assignment μ and the interpretation

```
boolean MATHSAT(formula φ, interpretation & I)
    μ = ∅;
    return MATHDPLL(φ, μ, I);

boolean MATHDPLL(formula φ, assignment & μ, interpretation & I)
    if (φ == ⊤) {                                   /* base       */
        I = MATHSOLVE(μ) ;
        return (I ≠ Null) ; }
    if (φ == ⊥)                                     /* backtrack */
        return False;
    if {a literal l occurs in φ as a unit clause}   /* unit propagation   */
        return MATHDPLL(assign(l, φ), μ ∪ {l}, I);
    if Likely-Unsatisfiable(μ)                      /* intermediate assignment check */
        if not MATHSOLVE(μ)
            return False;
    l = choose-literal(φ);                          /* split       */
    return    (MATHDPLL(assign(l, φ), μ ∪ {l}, I)  or
               MATHDPLL(assign(¬l, φ), μ ∪ {¬l}, I) );
```

Figure 1: Pseudo-code of the basic version of the MATHSAT procedure.

$\mathcal{I}$. MATHDPLL tries to find a truth assignment μ propositionally satisfying φ which is satisfiable from the mathematical viewpoint. Basically, MATHDPLL is a variant of DPLL, modified to work as an enumerator of truth assignments, whose satisfiability is recursively checked by MATHSOLVE. (The function *assign*(l, φ) assigns l to $\top$ in φ and propositionally simplifies the result.) The key difference w.r.t. standard DPLL is in the "base" step. Standard DPLL needs to find only one satisfying assignment μ, and thus simply returns $True$. MATHDPLL instead also needs to check the satisfiability of μ, and thus it invokes MATHSOLVE(μ). Then it returns $True$ if a non-null interpretation satisfying μ is found; it returns $False$ and backtracks otherwise. MATHSOLVE takes as input an assignment μ and returns either an interpretation $\mathcal{I}$ satisfying μ or $Null$ if there is none. In our implementation, MATHSOLVE first performs all the substitutions allowed by the equalities in μ. Then, if only inequalities with two variables are left, then a variant of Bellman-Ford minimal path algorithm is invoked, a linear programming procedure otherwise. Notice that MATHSAT works in polynomial space. In [ABC+02] some improvements to the procedure of Figure 1 are described (e.g. preprocessing and sorting, intermediate assignment checking, triggering, math-driven backjumping and learning) which we have implemented.

3 MATHSAT version 2: advances

In the following we will denote by MATHSAT.1 the version of MATHSAT of [ABC+02, ACKS02, ABCS03] and by MATHSAT.2 the current version.

3.1 A new SAT solver

A major bottleneck for MATHSAT.1 is the underlying SAT solver SIM [GMTZ01], which does not implement many useful state-of-the-art techniques. To overcome this problem, MATHSAT.2 is instead built on top of MINISAT [ES03]. MINISAT is a Chaff-like [MMZ+01] SAT solver featuring advanced techniques such as watched literals for value propagation, the VSIDS splitting heuristic, efficient conflict-driven backtracking and learning, and restarts. MINISAT com-

bines efficiency with simplicity, as it is well-engineered and it has a very clear, well-documented API, and is thus easy to modify and customize to one's needs.

3.2 Static Learning

A general problem of the MATHSAT approach is that the SAT solver may possibly enumerate a large number of assignments which are intrinsically inconsistent in the underlying theory.

On some specific kinds of satisfiability problems, like that of disjunctive temporal constraints problems (DTP) [ACG99], it is possible to easily detect a priori pairs of theory literals which are intrinsically mutually inconsistent (like, e.g., $\{(x - y \leq 3), (y - x \leq -2)\}$ and $\{\neg(x - y \leq 3), (x - y \leq 2)\}$). If so, binary mutual exclusion clauses (e.g., $\neg(x - y \leq 3) \vee \neg(y - x \leq -2)$ and $(x - y \leq 3) \vee \neg(x - y \leq 2)$) can be added a priori to the formula, which allow the solver to avoid generating any assignment including the inconsistent pairs. This technique, called $IS(2)$, allows for dramatically reducing the computational effort of solving DTP tests [ACG99]. Thus, to improve our competitiveness on these kinds of problems, we've implemented and tested $IS(2)$ for MATHSAT.2.

$IS(2)$ is a particular form of "static learning": in a preprocessing phase, constraints on the theory atoms of the formula can be "learned" and added in the form of clauses to the input formula, which guide the SAT solver to avoid generating theory-inconsistent assignments. (For instance, we embedded some of such techniques in our encoder for MATHSAT-based Bounded Model Checking for Timed and Hybrid automata [ACKS02, ABCS03].) We are currently implementing other forms of static learning, for more expressive theories, in future versions of MATHSAT.

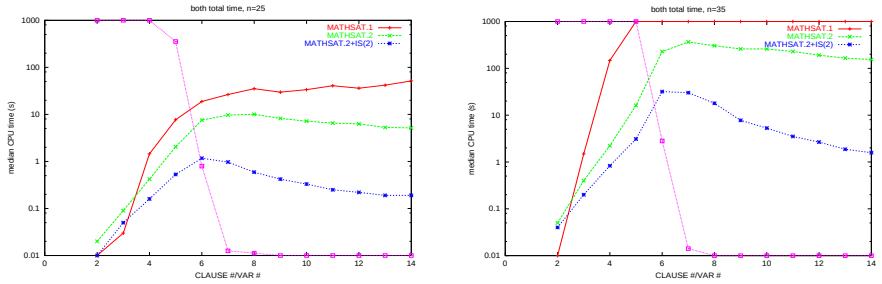
Notice that, unlike the extra clauses added in [SSB02, SLB03], the clauses added by static learning refer only to atoms which occur in the original formula, that is, no new atom is added. This means that the boolean search space is not enlarged. Furthermore, we notice that these added clauses are not needed for completeness, but they are only used for efficiency, in order to prune the search space.

3.3 Math Incrementality

Another major bottleneck of MATHSAT.1 is the fact that the mathematical solver is repeatedly called to check the consistency of intermediate assignments, which are increasingly large (i.e. μ, $\mu \cup \mu'$, $\mu \cup \mu' \cup \mu''$), as long as they are found satisfiable. If the mathematical solver is a memoryless subroutine (i.e. a function of the current assignment being checked), each call restarts the computation from scratch, and is likely to repeat the same partial computations over and over.

Therefore, in MATHSAT.2 we have implemented an *incremental*, version of the mathematical solver, with a stack-based interface that mimicks the behaviour of the stack-based boolean search. This solver remembers the computation status from one call to the other and, at each call, only the new math literals are added to the assignment; the mathsolver is only requested to normalize the new constraints, restarting from the saved status.

The core of the implementation is the CASSOWARY linear solver, a LP tool based on an incremental version of the simplex algorithm [BB01]. Besides being incremental, CASSOWARY, unlike LPSOLVE, is also amenable to mathematical learning: when given an inconsistent set of (in)equalities, it returns a clear indication of the subset of (in)equalities which caused the incon-



25 vars 35 vars

Figure 2: Comparison between MATHSAT.1, MATHSAT.2 and MATHSAT.2+$IS(2)$ on random DTP tests. $k = 2$, $\#vars = 25, 35$, $\#clauses/\#vars$ in $[2, ..., 14]$. 100 sample formulas per point. Median CPU times (secs). Background: satisfiability rate. (To save CPU time, a test is stopped whenever 50% of the samples exceed the timeout of 1000 seconds.)

sistency. This allows to construct (and lift at the boolean level, in the form of a clause) a conflict set, and results in an efficient implementation of mathematical backjumping and learning also for general-purpose linear (in)equalities.

We are currently investigating new techniques for combining math incrementality with the layered strategy described in [ABC+02].

4 Some experimental results

4.1 The test benchmarks

As a first application example, we consider the problem of solving the consistency of *disjunctive temporal problems* (DTP) proposed first in [SK98] and then used also in [ACG99, ABC+02]. Following [SK98], we encode DTP's as particular MATHSAT formulas in the form "$\bigwedge_i \bigvee_j (v_{1_{ij}} - v_{2_{ij}} \leq c_{ij})$", $v_{k_{ij}}$ and c_{ij} being real variables and integer constants respectively; a DTP is produced by randomly generating m distinct clauses of length k of the above form; 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. Some empirical tests performed on these benchmarks are presented in Figure 2.

As a second benchmark, we use 93 formulas taken from those used in [ABC+02, ACKS02, ABCS03], plus some other taken from equivalence checking of sequential circuit designs with bounded variables. Some empirical tests performed on these benchmarks are presented in Figure 3. Notice that these formulas are not restricted to difference constraints but, rather, may contain full linear expressions.

4.2 The role of the SAT solver

To emphasize the role played by the SAT solver used, we present first some tests comparing MATHSAT.1 and a first version of MATHSAT.2. In the latter, IS(2) and math incrementality are not implemented, so that substantially MATHSAT.2 here differs from MATHSAT.1 only on the fact that it uses MINISAT rather than SIM.

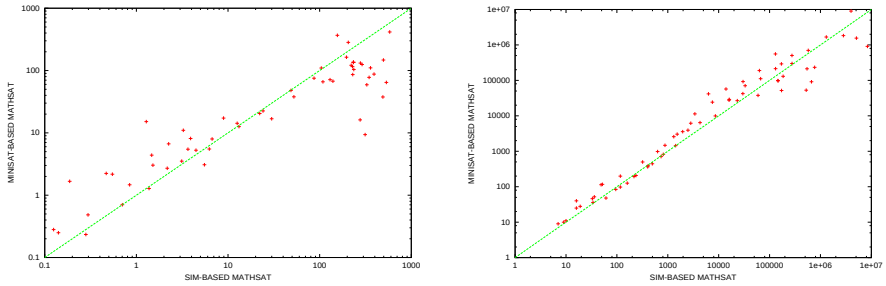


Figure 3: Comparison between MATHSAT.1 and MATHSAT.2 on a set of 93 formulas. Left: CPU time. Right: Size of the boolean search space.

First Consider the two upper plots on the random DTP tests of Figure 2. We notice that MATHSAT.2 outperforms MATHSAT.1 in both sets of testbenches. In particular, unlike MATHSAT.1, MATHSAT.2 is able to complete the whole benchmark for $n = 35$ without reaching the median value of 1000 seconds.

Second, consider the scattered plots on the 93 formulas of Figure 3. (Points above the diagonal line represent better results for MATHSAT.1, points below the diagonal line represent better results for MATHSAT.2.) Looking to the left plot (CPU times) we notice that, although MATHSAT.2 does not always win, it typically outperforms MATHSAT.1 on the harder benchmarks, with a performance gap of more than one order of magnitude on some instances. If we analyze in particular the size of the boolean search space (that is, the number of elementary assignments of truth values to atoms), we notice an analogous behaviour, although the gaps are reduced in both directions.

4.3 The role of static learning

To highligh the effectiveness of $IS(2)$ in DTP, we consider the two lower plots in both tests of Figure 2. We notice that MATHSAT.2+$IS(2)$ outperforms MATHSAT.2 up to two orders of magnitude, and that the performance gap increases with the size of the formulas and with the number of variables.

4.4 The role of incrementality

Finally, in order to highlight the effectiveness of incrementality for the mathematical solver, we consider the scattered plots of Figure 4, in which we run different versions of MATHSAT.2 on a subset of the formulas of Figure 3 (we consider only formulas involving full linear mathematical atoms, in which the linear solvers are used). Given the fact that the incremental version of MATHSAT.2 based on the CASSOWARY LP solver is not layered, we have compared it with both a non-layered (left) and a layered (right) version of MATHSAT.2 based on LPSOLVE.

We notice that the (non-layered) incremental version outperforms both the non-layered (left) and the layered (right) non-incremental ones. If confirmed by more extensive experiments, this may suggest that, if needed, we may sacrifice some of the layering to the incrmentality issue.

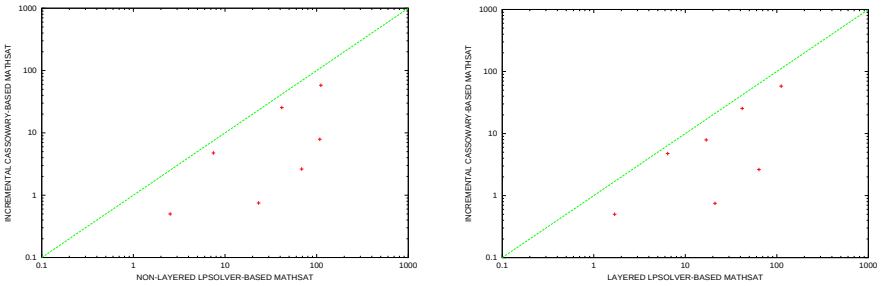


Figure 4: Left: Incremental CASSOWARY-based vs. Non-layered LPSOLVE-based MATHSAT.2. Right: Incremental CASSOWARY-based vs. Layered LPSOLVE-based MATHSAT.2.

5 Conclusions and Future Work

This extended abstract reports on the advances in the development of the MATHSAT decision procedure with respect to the version presented in [ABC+02]. We have discussed how the integration of a simple though efficient SAT solver results in a significant improvement. We have also shown that there is an interesting tradeoff between carrying out mathematical reasoning at the boolean level, and at the mathematical level: the idea of static learning allows to efficiently lift at the boolean level some mathematical information which could be discovered and learned (in a more costly manner) at run-time. Finally, we have shown the idea of incremental mathematical reasoning: using a mathematical solver (e.g. CASSOWARY [BB01]) which is able to extend the theory it is currently reasoning on has a clear advantage compared to using a memoryless mathematical reasoner.

We are currenlty working in several directions. The first one is to eliminate the limitation of MATHSAT.1, that works only with numerical variables in the domain of the Reals, since some applications requires handling integer variables. Whilst in separation logic handling integer values is rather straightforward (see, e.g., [SSB02]), the general linear case is more complicated and deserves some care. A naive way to extend MATHSAT to work in $\mathbb{Z}$ would be to simply substitute the linear solver of MATHSAT with an integer linear solver (e.g., Omega [Pug92], or a recent version of LPSOLVE [Ber99]). Unfortunately, it is well-known that LP in the integers is a much harder problem than that in the reals (see, e.g., [BW01]), and this solution would likely cause an unacceptable overhead. Therefore, we have identified the following "layered" solution: invoke an integer linear solver only when it is strictly necessary. We start from the observation that, in many practical problems, a set of constraints having a solution in $\mathbb{R}$ also has one in $\mathbb{Z}$. Thus, the idea is to run the integer linear solver in LPSOLVE [Ber99] on an assignment μ if and only if μ is a complete assignment (that is, μ propositionally satisfies the formula) and the real linear solver has verified it is consistent in $\mathbb{R}$. We don't run the integer solver in intermediate assignment checking calls: if an intermediate call to the real linear solver returns "satisfiable", we trust it and go on. Clearly, we might lose the benefit of early pruning of intermediate assignments that are satisfiable in $\mathbb{R}$ but not in $\mathbb{Z}$; on the other hand, however, we avoid the big overhead of invoking the integer linear solver at every intermediate check. We are also comparing MATHSAT with other solvers; the results will be reported in the final version of this extended abstract.

Directions of future work include the design of a new version of MATHSAT able to deal with uninterpreted functions, and with non-linear arithmetic.

References

[ABC+02] 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.*, 2002.

[ABCS03] 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.

[ACG99] A. Armando, C. Castellini, and E. Giunchiglia. SAT-based procedures for temporal reasoning. In *Proc. European Conference on Planning, CP-99*, 1999.

[ACKS02] 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, November 2002.

[BB01] G. J. Badros and A. Borning. The Cassowary Linear Arithmetic Constraint Solving Algorith. *ACM Transactions on Computer Human Interaction*, 8(4):267–306, december 2001.

[BDS02] C. Barrett, D. Dill, and A. Stump. Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT. In *14th International Conference on Computer-Aided Verification*, 2002.

[Ber99] Michel Berkelaar. The solver lp_solve for Linear Programming and Mixed-Integer Problems. 1999. Available at `http://elib.zib.de/pub/Packages/mathprog/linprog/lp-solve/`.

[BW01] A. Bockmayr and V. Weispfenning. Solving numerical constraints. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 12, pages 751–842. Elsevier Science, 2001.

[DLL62] M. Davis, G. Longemann, and D. Loveland. A machine program for theorem proving. *Journal of the ACM*, 5(7), 1962.

[dMRS02] Leonardo de Moura, Harald Ruess, and Maria Sorea. Lazy Theorem Proving for Bounded Model Checking over Infinite Domains. In *18th International Conference on Automated Deduction*, 2002.

[ES03] N. Een and N. Sorensson. An Extensible SAT-solver. In *Proc. SAT'03*, volume 2919 of *LNCS*. Springer, 2003.

[FJOS03] Cormac Flanagan, Rajeev Joshi, Xinming Ou, and James Saxe. Theorem proving using lazy proof explication. In *15th International Conference on Computer Aided Verification (CAV)*, Lecture Notes in Computer Science. Springer-Verlag, 2003.

[GMTZ01] E. Giunchiglia, M. Maratea, A. Tacchella, and D. Zambonin. Evaluating search heuristics and optimization techniques in propositional satisfiability. In *Proc. IJCAR-01*, 2001.

[MMZ+01] M. W. Moskewicz, C. F. Madigan, Y. Z., L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference*, 2001.

[Pug92] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 1992.

[SK98] K. Stergiou and M. Koubarakis. Backtracking algorithms for disjunctions of temporal constraints. In *Proc. AAAI*, pages 248–253, 1998.

[SLB03] S. A. Seshia, S. K. Lahiri, and R. E. Bryant. A Hybrid SAT-Based Decision Procedure for Separation Logic with Uninterpreted Functions. In *Proc. 40th Design Automation Conference (DAC)*, 2003.

[SSB02] O. Strichman, S. Seshia, and R. Bryant. Deciding separation formulas with SAT. In *Proc. of Computer Aided Verification, (CAV'02)*, LNCS. Springer, 2002.