# A SAT-based Decision Procedure for the Boolean Combination of Difference Constraints

Alessandro Armando[1], Claudio Castellini[1],
Enrico Giunchiglia[2], and Marco Maratea[2]

[1] AILab, DIST - University of Genova
viale Francesco Causa, 13 — 16145 Genova (Italy)
http://www.ai.dist.unige.it
{armando,drwho}@dist.unige.it

[2] STARLab, DIST - University of Genova
viale Francesco Causa, 13 — 16145 Genova (Italy)
http://www.star.dist.unige.it
{enrico,marco}@dist.unige.it

**Abstract.** The problem of deciding satisfiability of Boolean combinations of difference constraints is at the core of many important techniques such as planning, scheduling and bounded model checking of real-time systems. Efficient decision procedures for this class of formulas are, therefore, strongly needed. In this paper we present TSAT++, a system implementing a SAT-based decision procedure for this problem, and the techniques implemented in it; in particular, TSAT++ takes full advantage of recent SAT techniques. Comparative experimental results indicate that TSAT++ outperforms its competitors both on randomly generated, hand-made and real world problems.

## 1 Introduction

In temporal reasoning, one of the best known and studied formalisms is the so-called Simple Temporal Problem (STP) [DMP91], consisting of a conjunction of *difference constraints*, i.e., constraints of the form $x - y \leq c$ where $x$ and $y$ are variables ranging over a fixed numeric domain (typically the integers or the reals) and $c$ is a numeric constant.

STPs are tractable, but their expressiveness is rather limited; therefore, recently, several extensions to them have been introduced, allowing propositional atoms, disjunctions and negations of binary constraints. Such extensions are more expressive than the STP but retain most of its conciseness and clarity; still, they are in general able to express complex problems such as planning, scheduling and verification of real-time hardware.

It is then clear that efficient decision procedures for these extensions are strongly needed; in fact, in the last five years, at least six systems have been proposed that are able to deal with disjunctions of difference constraints, four of which in the AI literature and two in the formal verification literature, meaning that the topic is hot and interdisciplinary. Not surprisingly, five out of these six systems are SAT-based or CSP-based; this means that the satisfiability of a problem $\phi$ is determined by

1. (enumeration) generating a set of propositional atoms and difference constraints "propositionally satisfying" $\phi$, using SAT or CSP techniques, and
2. (satisfiability checking) testing the satisfiability of each generated set using standard techniques (such as, e.g., the Bellman-Ford procedure).

As we will see, the enumeration, and thus the specific SAT/CSP technique being used in the enumeration phase, is crucial to the efficiency of the procedure. Despite this, none of the aforementioned five systems take advantage of the recent developments in the SAT field.

In this paper we first survey the techniques and optimizations that have been realized in the aforementioned solvers, trying to highlight their pros and cons, meanwhile introducing our system TSAT++. TSAT++ is able to deal with arbitrary conjunctions, disjunctions, and negations of difference constraints and propositional atoms. We then briefly describe how we have integrated in TSAT++ most of the above mentioned techniques, plus some new ones. Lastly we show the outcome of an extensive comparative analysis among the solvers, including TSAT++.

TSAT++ integrates the latest techniques proposed in the SAT field (and, in particular, those proposed in [MMZ+01]) and uses new ideas designed to take maximum advantage from the techniques used in the enumeration phase. Thanks to these methods and their fruitful integration, we show that TSAT++ has a clear edge over its competitors; in fact, the analysis shows that TSAT++ is

- at least 2 orders of magnitude faster in the hard region on real-valued randomly generated problems;
- at least 6 times faster in the hard region on integer-valued randomly generated problems;
- on average at least a factor of 4 faster on instances coming from real world problems; and
- up to 3 orders of magnitude faster on hand-made problems

*than its fastest competitor in each category*. These results are even more interesting if one considers that, unlike most of the competitors, TSAT++ is not tuned nor customized on any particular class of problems.

The paper is structured as follows: after some preliminaries (Section 2), we introduce TSAT++ and describe the techniques implemented in it (Section 3); then we show our experimental results (Section 4) and finally draw some conclusions.

## 2 Preliminaries

### 2.1 Temporal Reasoning Problems

Let $D$ be either the set of the integer or real numbers. Let also $\mathcal{V}$ and $\mathcal{P}$ be two disjoint sets of symbols, called *variables* and *propositional atoms* respectively. Then a *difference constraint* is an expression of the form $x - y \leq c$ where $x, y \in \mathcal{V}$ and $c$ is a numeric constant.

An *atom* is either a difference constraint or a propositional atom; a *literal* is either an atom or its negation (if $a$ is an atom, then $\overline{a}$ abbreviates $\neg a$ and $\overline{\neg a}$ stands for $a$); and lastly, a *Temporal Reasoning Problem* (TRP) is a Boolean combination of literals.

A *TRP-assignment* (on $D$), or *assignment* when it is not ambiguous, is a function mapping each variable to an element of $D$, and each propositional atom to the truth values $\{\bot, \top\}$. An assignment $\sigma$ is extended to map a TRP to $\{\bot, \top\}$ by defining

- $\sigma(x - y \leq c) = \top$ if and only if $\sigma(x) - \sigma(y) \leq c$, and
- $\sigma(\phi) = \top$ (with $\phi$ being a TRP) according to the truth tables of propositional logic.

Let $\phi$ be a TRP. We say that an assignment $\sigma$ *satisfies* $\phi$ if and only if $\sigma(\phi) = \top$; such an assignment will be called a *TRP-model*, or *model* when it is not ambiguous, of $\phi$. A TRP is *satisfiable* (in $D$) if and only if there exists an assignment (on $D$) which satisfies it. A finite set of literals is satisfiable (in $D$) if and only if their conjunction, as a TRP, is.

Here we deal with the problem of determining whether a TRP is satisfiable or not, having fixed the domain of interpretation $D$. Clearly the problem is NP-complete, no matter which of the above mentioned sets $D$ is.

In the following, we will use the term *valuation* to mean a mapping from atoms to $\{\bot, \top\}$, extended to arbitrary TRPs according to the truth tables of propositional logic; a valuation *satisfies* a TRP if and only if it makes the TRP true. On the other side, we will represent a valuation as the set of literals in it assigned to true; then an *(un)satisfiable* valuation is such that the set of difference constraints represented by the associated literals is (un)satisfiable.

Also, we restrict our attention to TRPs in CNF. This is not a limitation since any TRP can be efficiently reduced to an equi-satisfiable formula in CNF. With this assumption, we represent a TRP as a set of clauses, each clause being a set of literals. As is customary, clauses with one literal only will be called *unit clauses*.

TRPs actually represent an interesting extension to the STP, introducing disjunction and negation of difference constraints and propositional atoms; this added value makes them as expressive as Separation Logic [SSB02], which employs the predicates $<$, $\geq$, $>$, $=$, and $\neq$. Another well-known related framework, the Disjunctive Temporal Problem (DTP), is limited to two disjuncts per clause and admits no propositional atoms, but it is as expressive as a TRP (introducing new variables). In general, valuations of TRPs, restricted to difference constraints, can be seen as Simple Temporal Problems — they are equivalent to conjunctive sets of difference constraints.

### 2.2 SAT and CSP-based procedures

Beside TSAT++, the systems we will be considering here are: the system presented in [SK98] (that we will call SK), Tsat [ACG00], CSPi [OC00], Epilitis[TP03], SEP [SSB02] and MathSAT [ABC$^+$02]. As far as we know, these systems represent a fair snapshot of the current state-of-the-art; and, with the exception of SEP (see [SSB02] for more details), all approaches and systems built so far for TRPs and similar problems are quite alike from an algorithmic point of view. In fact, given a TRP $\phi$, they all work by

1. (enumeration) generating all the valuations $\mu$ which satisfy $\phi$,
2. (satisfiability checking) for each $\mu$, testing whether it is satisfiable.

Enumeration can be done as search in a Constraint Satisfaction Problem (CSP) associated to the basic temporal reasoning problem (systems which do this are SK, CSPi, Epilitis) or by solving the corresponding SAT problem (Tsat, MathSAT, TSAT++). In the first approach, search is performed in a meta-search space in which a new variable is associated with each clause, its domain being the set of disjuncts in the clause. In the SAT-based approach, the given TRP is abstracted to a propositional formula obtained by substituting each distinct binary constraint with a newly introduced propositional atom.

As one can see, SAT- and CSP-based approaches are tightly connected, and it is therefore not surprising that in their basic versions and starting from Tsat, all the systems perform the following steps:

1. assign to true the literals in unit clauses;[1]
2. if there are no more literals to assign according to the previous step, they branch on a literal $l$ (i.e., assign true to $l$), and, upon failure of the subsequent search, add the negation of $l$ to the current state and continue the search, till either a satisfying assignment is found, or backtrack occurs.

The similarity to the search performed by SAT solvers is apparent; despite this, none of the above systems incorporates the last advancements done in the SAT field.

As far as expressiveness is concerned, TSAT++ is able to deal with any TRP; SK, Tsat, CSPi and Epilitis are restricted to DTPs; SEP is as expressive as TSAT++, and neither is comparable to MathSAT, since MathSAT allows for arbitrary linear constraints as atoms and does not allow the integers to be considered as domain of interpretation.

## 3 TSAT++

In this section we describe the main ideas behind TSAT++. Most of the terminology we will be using from now on is customary in the AI and Formal Methods literature. Rather than as a monolithic system, TSAT++ is conceived as an open platform for the implementation of such ideas and techniques. For a more detailed description of the system, refer to [ACG+04].

In particular, we will describe $(i)$ the computation done before the search starts (*preprocessing*), $(ii)$ the way the search space is pruned after each branching node (*look-ahead*), $(iii)$ the way recovery from failures happens (*look-back*); $(iv)$ the heuristics used for picking the literal on which to branch (*branching rule*), and $(v)$ the procedure used for checking the satisfiability of a set of literals (*satisfiability checking*).

TSAT++ employs an API-like modified version of SIMO [GMT03] for the enumeration phase.

[1] In CSPi and Epilitis, priority of unit clauses is embedded in the heuristics used for selecting the literal to branch on. Furthermore, these systems employ *forward checking*, which removes the binary constraints whose negation is entailed by the current valuation.

### 3.1 Pre-processing

One drawback of the generate-and-test approach is that (exponentially) many trivially unsatisfiable valuations can be generated and then discarded, essentially because the machinery in charge of the enumeration phase knows nothing about difference constraints. This may happen both in SAT and CSP-based approaches. For example, in the enumeration phase there is no constraint relating the truth values of, e.g., $x - y \leq 3$ and $x - y \leq 5$. Thus, many trivially unsatisfiable valuations (e.g., with $x - y \leq 3$ assigned to true and $x - y \leq 5$ to false) can be generated.

In order to reduce this effect, in TSAT++ for each pair $c_1, c_2$ of difference constraints in the same variables and occurring in the input formula, the satisfiability of all possible pairs of literals built out of them, i.e., $\{c_1, c_2\}$, $\{\neg c_1, c_2\}$, $\{c_1, \neg c_2\}$, and $\{\neg c_1, \neg c_2\}$, is checked.

Assuming, e.g., $\{c_1, c_2\}$ is unsatisfiable, the clause $\{\neg c_1, \neg c_2\}$ is added to the input formula *before the search starts*. In our example, we would add the clause $\{\neg x - y \leq 3, x - y \leq 5\}$.

This can dramatically speed up the search, especially on randomly generated problems. In fact, e.g., as soon as $x - y \leq 3$ is assigned to true, $x - y \leq 5$ gets also assigned to true by unit propagation. This technique is an extension of a technique called IS(2), introduced in [ACG00].

### 3.2 Look-ahead

Consider a TRP $\phi$ and let $S$ be the set of literals assigned to true so far. The idea behind look-ahead techniques is to try to detect new literals $l$ that are *entailed* by $\phi$ and $S$, i.e., such that $l$ is satisfied by each assignment satisfying $\phi$ and $S$. If $l$ is one of such literal, we can $(i)$ add $l$ to $S$ and $(ii)$ simplify $\phi$ on the basis that $l$ is true. This has the beneficial effect of postponing the branching phase and in doing so it may lead to huge savings.

The basic look-ahead technique common to all solvers is unit-propagation. A simple profiling of the code of TSAT++ on real world problems reveals that most of the CPU time is spent in the enumeration phase (often more than 80%, sometime close to 100%), within which most of the time is spent by unit-propagation ($> 90\%$ in most cases). Therefore, the choice of a good data-structure for unit-propagation is capital.

*Two-literal watching* is an efficient data-structure for unit-propagation (see, e.g., [MMZ$^+$01]). With it, each clause maintains two fields meant to store two "watched" open (i.e. not assigned) literals. Assigning an atom and detecting new units, causes the visit of a sub-linear (in the number of occurrences of the atom) number of clauses. Further, following the same paper, when backtracking occurs, nothing needs to be undone, and thus backtracking takes constant time. On the other hand, by using standard *counters structures* as in, e.g., Tsat and MathSAT, assigning an atom and detecting new units has a cost which is at least linear in the number of occurrences of the atom. Furthermore, when backtracking occurs and an atom is unassigned, each operation needs be undone and this, again, has a cost which is linear in the number of occurrences of the atom. This alternative approach has a higher computational cost than two-literal watching, especially when managing large clauses, such as those emerging from learning.

Still, even in SAT, it is not always true that two-literal watching is better than counters structures. Indeed, the former are a de-facto standard for SAT solvers designed for real world applications, while the latter are at the basis of the most efficient solvers on randomly generated problems [BS03].

Having in mind real world applications, we implemented two-literal watching in TSAT++: for these applications, look-back mechanisms based on learning are fundamental, and these imply handling large clauses (with hundreds of literals). It turns out, however, that TSAT++ outperforms the other solvers even on random problems.

Unit-propagation prunes valuations on the basis of propositional reasoning. However, it may be the case that the set of literals $\mu$ assigned at a certain point of the search tree, is already unsatisfiable. In this case, there is no point in continuing the search expanding $\mu$. Because of this, before each branching we may check whether $\mu$ is satisfiable or not. If not, we can immediately force backtracking. This technique is called *early-pruning* and it is implemented by all the solvers we considered.

### 3.3 Look-back

If recovery from a failure is performed by simple chronological backtracking, it is not infrequent to keep exploring a possibly large subtree whose leaves are all dead-ends, especially if the failure is due to some choices performed way up in the search tree. The solution to this problem is to jump back over the choices that do not belong to the reason for the failure. Intuitively, if $S$ is a set of literals such that $S \cup \phi$ (where $\phi$ is the input CNF formula) is unsatisfiable, then a *reason* $R$ for $S$ is a subset of $S$ such that $\phi \cup R$ is unsatisfiable. Reasons are initialized as soon as an unsatisfiability is detected, and updated while backtracking. The corresponding technique is known as *(Conflict-Directed) Back-jumping* (CBJ) [Pro93].

With *learning* (see [BM96,SS99]), each reason $R$ computed while back-jumping is turned into the clause $\{\overline{l} \mid l \in R\}$ that may be added to the input formula. Learned clauses will prune the subsequent search space, thus avoiding the repetition of the same mistakes. On the other hand, exponentially many reasons can be learned, and each learned clause causes an overhead when assigning literals. In practice it is necessary to introduce criteria $(i)$ for limiting the clauses that have to be learned and $(ii)$ for removing some of them.

TSAT++ features *1-UIP learning* [MMZ$^+$01]. This technique ensures that at each decision level of each branch at most one clause is added to the input formula. Still, an exponential blow-up may happen. To prevent this in TSAT++, each added clauses is analyzed with a given periodicity and (possibly) deleted. Standard alternatives to 1-UIP learning are [BM96]

1. *relevance-bounded learning* of order $n$ (used in MathSAT with $n = 3, 4$) and
2. *size-bounded learning* of order $n$ (used in Epilitis with $n = 10$).

Compared to the 1-UIP learning implemented in TSAT++, both MathSAT and Epilitis may store more than one clause per level.

### 3.4 Branching rule

TSAT++ uses a conflict-based heuristic, whose basic idea is to select the literal mostly occurring in the most recently learned clauses. The rationale behind it is that learned clauses represent conflicts among the literals that have emerged during the search. By satisfying these clauses we avoid doing the same "mistake" over and over again. However, not all the learned clauses are equally important: Indeed, some of them, e.g., those discovered at the beginning of the search, may become obsolete for guiding the search in the current branch. Thus, the score associated with each literal is periodically divided by 2, giving more relevance to the atoms that will occur in the newly discovered conflicts.

Of course, such conflict-based heuristics make sense only for solvers with learning. Epilitis uses a similar heuristics. The main difference is that, in Epilitis, all conflicts are equally important, i.e., it does not focus on the atoms in the most recently learned clauses. MathSAT employs a wide variety of heuristics, some of which specifically designed for solving a specific class of problems. However, even though MathSAT uses learning and thus could employ a conflict-based heuristic, all its heuristics are MOMS-based (Maximum Occurrences in clauses of Minimal Size): They give higher scores to literals in shorter clauses. These heuristics have been mutuated from the SAT literature, and are used also by Tsat. In the CSP-based systems, MOMS-based heuristics correspond to the Minimum Remaining Value (MRV) heuristics, used in SK and CSPi.

It is not easy to compare these heuristics for TRPs. One could make some considerations on the basis of what is known in the SAT literature. For instance, in SAT, MOMS-based heuristics are known to be better than conflict-based heuristics on randomly generated problems. However, it is not clear whether such considerations still hold in this setting. For instance, the results in [TP03] point out that its conflict-based heuristics is better than its MOMS-based heuristics even on randomly generated problems. What is true is that MOMS-based heuristics are not compatible with a two literal watching data structure, since they require to know which clauses are active and their length. [2]

### 3.5 Satisfiability checking

Consider a set $S$ of literals. For all the procedures here considered, an effective method for checking the satisfiability of $S$ is needed. Moreover, when $S$ is unsatisfiable, it is important to be able to extract a reason of its unsatisfiability, i.e., an unsatisfiable subset $S'$ of $S$. Of course, a naïve selection of such a set $S'$ is the set $S$ itself; however, applying this selection is seldom a good idea since $S'$ is to be used by the look-back mechanisms, e.g., to backjump over irrelevant nodes. It is thus of fundamental importance to keep $S'$ as "small" as possible in order to try and maximize the benefits of the look-back.

We now describe how we compute such a small set $S'$. For the time being, let us assume that $S$ is just a set of difference constraints, i.e., that we are facing a STP. We

[2] One could argue that two literal watching structures are compatible with unit-based heuristics [LA97]. However, as a preliminary step, unit-based heuristics use a MOMS-based criteria in order to select the variables to score with a unit-based heuristics.

will see later how to generalize the discussion to arbitrary literals. The standard method to check the satisfiability of a STP $S$ is the Bellman-Ford procedure (BF — see, e.g., [CLR98]). The basic idea is to associate with $S$ a *constraint graph*, whose nodes are the variables in $S$, and which has an edge from $y$ to $x$ with weight $c$, for each constraint $x - y \leq c$ in $S$. Then, an extra node $s$ (the "source") connected to all the other nodes with weight 0 is added, and BF is used to compute the "single source shortest-paths" problem. If $S$ is satisfiable, there are no negative cycles in the graph, and BF returns true; otherwise, a minimal (under a suitable set ordering, the simplest of which being cardinality), unsatisfiable subset $S'$ of $S$ can be easily detected by inspection of the constraint graph.

Notice that the constraint graph of $S$ may have several different negative cycles, each one corresponding to a minimal unsatisfiable subset of $S$. The standard approach amounts to stopping the search as soon as one such negative cycle is detected. TSAT++ instead continues the search in order to determine a negative cycle involving the smallest number of nodes (corresponding to an unsatisfiable set with minimal cardinality). This modification does not alter the overall complexity of BF, which remains $O(n \times m)$, where $n$ and $m$ are the numbers of variables and constraints in $S$ respectively. On the other hand, the advantage of returning an unsatisfiable set with fewer constraints is that it possibly leads to pruning a larger portion of the search space.

Furthermore, when $S$ is a valuation satisfying the input TRP $\phi$, some of the literals in $S$ may be not necessary to satisfy $\phi$. In other words, there may be a literal $l$ in $S$ such that, for each clause $C \in \phi$ with $l \in C$, there is another literal $l'$ in $S \cap C$. If this is the case, also $(S \setminus \{l\}) \cup \{\bar{l}\}$ satisfies $\phi$, and we can safely check the satisfiability of $S \setminus \{l\}$ instead of $S$. TSAT++ may recursively eliminate such literals $l$ from $S$. If $S' \subseteq S$ is the resulting set, it will then check the satisfiability of $S'$. We call the above procedure *reduction*, and it may be useful because

- if $S$ is satisfiable, so is $S'$, and we are done;
- if $S$ is unsatisfiable, it may nevertheless be the case that $S'$ is satisfiable, and we interrupt the search and exit with a satisfying assignment;
- if $S$ and $S'$ are both unsatisfiable, checking the satisfiability of $S$ instead of $S'$ can cause exponentially many more satisfiability checks. In fact, any valuation extending $S'$ also satisfies $\phi$, and each could be generated and then rejected by TSAT++.

The last two cases are of particular relevance in TSAT++. In fact, because of the two-literal watching data structure, the generated valuations satisfying $\phi$ are always total. Thus, it is very often the case that huge portions of the difference constraints in $S$ are irrelevant for satisfying $\phi$ and, by removing them, we end-up in a set $S'$ with many less difference constraints. Notice that the reduction procedure is not to be applied when early pruning is enabled. With early pruning, the hope is that $S$ is unsatisfiable in order to stop the search. If $S'$ turns out to be satisfiable, we cannot conclude about $S$, and we have to go on expanding $S$.

So far, we have been using the assumption that $S$ is a set of difference constraints. The problem is how to deal with the negation of difference constraints. Assume we have $\neg x - y \leq c$ in $S$. Then, such a literal is equivalent to $y - x < -c$, and we can replace every such constraint in $S$ with a constraint $y - x \leq d$, where $d$ is

– the maximum integer strictly smaller than $-c$, if variables range over the integers; and
– $-c - \frac{1}{10^p(n^2+1)}$, otherwise. In the expression, $n$ is the number of variables in $S$, and $p$ is the maximum number of digits appearing to the right of the decimal point (assuming that there are no useless "0"), in any of the constants of the input TRP. If all the constants are integers, $p = 0$.

The resulting set does not contain any negation of difference constraint, and it is satisfiable if and only if the initial set is (this follows from Theorem 3 in [GC97]).

## 4 Experimental Analysis

### 4.1 Experimental setting

In order to thoroughly compare TSAT++ with the state-of-the-art we have considered a wide variety of publicly available random, real world and hand-made TRPs (the classification has been done following what is standard practice in the SAT competition [BS03]). As far as the solvers are concerned, we have initially considered all the publicly available systems, that is, the above mentioned SK, CSPi, Epilitis, MathSAT, SEP and Tsat plus, of course, TSAT++. After a first run, we have discarded SK, because it is clearly not competitive with respect to the others.

Each solver has been run on all the benchmarks it can deal with, not only on the benchmarks the solver was analyzed on by the authors. In particular, Epilitis can only handle DTPs with binary clauses and integer valued variables; CSPi and Tsat can only handle DTPs with real valued variables; MathSAT can handle arbitrary TRPs with real valued variables; SEP and TSAT++ can handle arbitrary TRPs.

Each solver has been run using the settings or the version of the solver suggested by the authors for the *specific* problem instances; when not publicly available, we directly asked the authors for the best setting. TSAT++ has many possibilities, also beyond those described in this paper. Of the features described in this paper, only preprocessing, early pruning and reduction of satisfying assignments can be enabled and disabled at the command line.

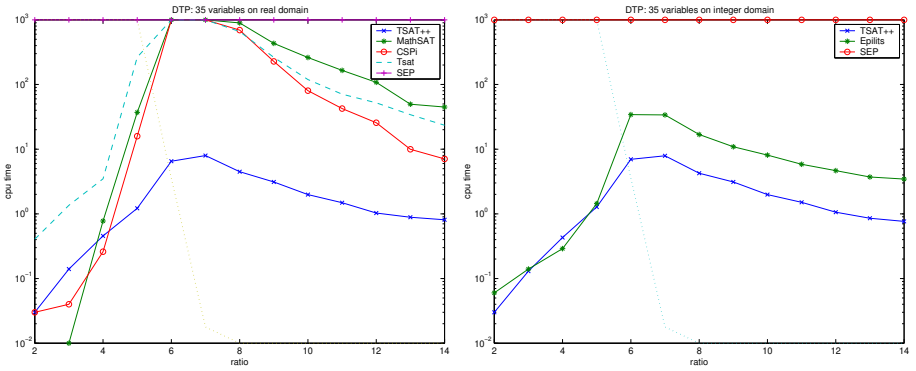
All experiments were run on a Pentium IV 2.4GHz processor with 1GB of RAM. CPU time is given in seconds; timeout was set to 1000 seconds.

### 4.2 Comparative evaluation on random DTPs

We start our analysis considering randomly generated DTPs as introduced in [SK98] and since then used as a benchmark in [ACG00,OC00,ABC+02,TP03]. DTPs are randomly generated by fixing the number $k$ of disjuncts per clause, the number $n$ of arithmetic variables, a positive integer $L$ such that all the constants are taken in $[-L, L]$. [3] Then, $(i)$ the number of clauses $m$ is increased in order to range from satisfiable to unsatisfiable instances, $(ii)$ for each tuple of values of the parameters, 100 instances are

[3] A random DTP generator is available at the URI of the SMT-LIB initiative, `http://goedel.cs.uiowa.edu/smtlib`.

generated and then given to the solvers, and $(iii)$ the median of the CPU time is plotted against the $m/n$ ratio. The results for $k = 2, L = 100$ and $n = 35$ are given in Figure 1: Plots (a) and (b) shows the performance when the variables are real and integer valued respectively.




(a) (b)

**Fig. 1.** Comparative analysis on (a) randomly generated DTPs with 35 real valued variables (b) randomly generated DTPs with 35 integer valued variables. Systems are stopped after 1000 seconds. Back: satisfiability percentage.

When $m/n \geq 6$, TSAT++ clearly outperforms all other systems: In the peak region, the solver that is closer to TSAT++ in this domain, namely Epilitis, is a factor of 6 slower on 35 variables (cf. plot (b)). This is a very positive result, taking into account that Epilitis only works on DTP with $k = 2$, and it has been thoroughly tested and optimized on this type of problems (see [TP03]).

All other systems are about 2 orders of magnitude slower than TSAT++ in the peak region. TSAT++ has been run with early pruning and pre-processing enabled, and these are fundamental for its performance on this test set: Without early pruning (resp. pre-processing) TSAT++ on the peak is slower by 2 (resp. 1) orders of magnitude. The fact that these two techniques are important comes at no surprise, and confirm previous results in [ACG00]. The new look-ahead, heuristics and look-back mechanisms used by TSAT++ explain the 2 orders gap with respect to Tsat.

Lastly, in order to evaluate the scalability of TSAT++, we tested it against DTPs with a larger number of variables, namely 40, 45, and 50; as far as we know, the larger problems tackled so far in literature have 35 variables (see [TP03]). The results (including also the results for 35 variables) are plotted in Figure 2. The plots show that TSAT++ does not timeout even with 50 variables and that it pays around a factor 4 on the hardest point when adding five variables. All other systems we considered were not able to deal with problems with 50 variables; also, the performance gap between TSAT++ and the other systems increased with the number of variables.

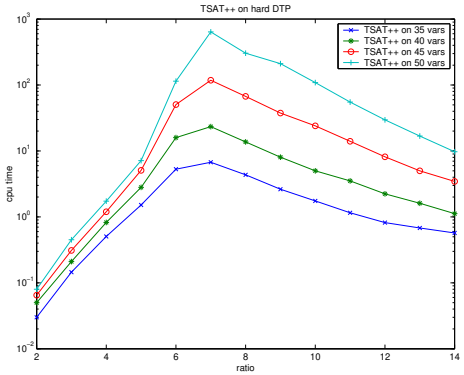



**Fig. 2.** Scalability of TSAT++. Curves represent TSAT++'s performance on DTPs with 35, 40, 45 and 50 real valued variables.

### 4.3 Comparative evaluation on real world problems

In this Subsection we consider

1. the 40 post-office benchmarks introduced in [ABC+02], coming in 4 series (consisting of 7, 9, 11, and 13 instances respectively) of increasing difficulty, and
2. the 16 hardware verification problems from [SSB02], 9 (resp. 7) of which are with real (resp. integer) valued variables.

The post-office benchmarks represent bounded model checking for timed automata; the hardware verification suite include scheduling, cache coherence protocol, load-store unit and out-of-order execution problems.

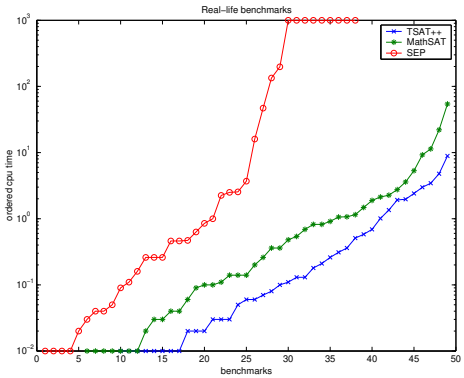



**Fig. 3.** Comparative analysis on real world problems. Systems are stopped after 1000 seconds.

Consider Figure 3. By looking at the results of MathSAT, SEP and TSAT++ on the post-office problems, our first observation is that SEP is not competitive on these problems: On 13 of the hardest instances, SEP had a segmentation fault in 11 cases, and on the other 2 hardest instances SEP is outperformed by orders of magnitude by TSAT++ and MathSAT. Our second observation is that TSAT++ (with pre-processing and assignment reduction) performs better than MathSAT up to a factor of 6, on *each single instance*: This is particularly remarkable given that the authors have customized a version of MathSAT explicitly for this kind of problems. Without pre-processing TSAT++'s performance is worse of about a factor of 3 and of about 10% worse without reduction.

Considering the hardware verification problems, all of them are easy to solve (less than 3s) for all the three solvers, except for SEP that timeouts on one instance. Of the 9 (resp. 16) runs of MathSAT (resp. SEP and TSAT++), only 3 take more than 0.1s. These observations are confirmed by Figure 3, which gives the overall picture of the results for MathSAT, SEP and TSAT++ on the 49 instances with real valued variables: The $x$-axis is the number of instances solved by each solver within the CPU time specified on the $y$-axis.

### 4.4 Comparative evaluation on hand-made problems

Finally, we consider the "hand-made" diamonds problems shown in [SSB02]. Given a parameter $D$, these problems are characterized by an exponentially large ($2^D$) number of satisfying valuations, some of which correspond to TRP-models; hard instances with a unique TRP-model can be generated. A second parameter, $S$, is used to make TRP-models larger, further increasing the difficulty. Variables range over the reals.

**Table 1.** Diamonds problems.

| D | S | unique | TSAT++ | TSAT++$^p$ | SEP | SEP (no c.m.) | MathSAT |
|---|---|---|---|---|---|---|---|
| 50 | 4 | N | 0 | 0.02 | 0.03 | 0.12 | 0.05 |
| 50 | 4 | Y | 0.01 | 0.14 | 0.84 | 0.07 | TIME |
| 100 | 5 | N | 0.01 | 0.11 | 0.13 | 1.18 | 0.61 |
| 100 | 5 | Y | 0.04 | 7.57 | 10.20 | 0.17 | TIME |
| 250 | 5 | N | 0.08 | 0.76 | 0.95 | 52.20 | 5.4 |
| 250 | 5 | Y | 0.21 | 194.99 | 288.30 | 0.77 | TIME |
| 500 | 5 | N | 0.29 | 4.46 | 5.92 | 742.99 | 21.22 |
| 500 | 5 | Y | 1.05 | TIME | TIME | 4.85 | TIME |

Table 1 shows comparative results on the diamonds problems. The third column denotes whether the problem has a unique TRP-model; the remaining columns show CPU times for TSAT++ with reduction of assignment enabled, TSAT++ plain version (denoted as TSAT++$^p$), SEP with and without conjunction matrix and MathSAT.[4]

[4] The configurations employed were suggested by the authors of SEP and MathSAT.

TSAT++ with reduction of assignment enabled clearly performs best, often by orders of magnitude; instances with a unique solution are more difficult than non-unique ones, as expected, except for SEP without conjunction matrix.

For this test set, the good interplay between look-back and satisfiability checking engines is fundamental. In particular, the reduction of assignment step is crucial, and this is clear from the comparison between TSAT++ and TSAT++$^p$ columns in Table 1: Without reduction, TSAT++ performs significantly worse, up to the point that problems that are solved in 1 second, are not solved without reduction within the time limit.

*Related Work.* Two further systems which tackle the same problem are LPSAT and CPLEX. LPSAT ([WW99]) has been excluded from the comparative analysis since it is algorithmically quite similar to MathSAT and it employs the SAT solver RelSat ([BS97]), similar to SIM, which is used in MathSAT. Therefore we do not expect LPSAT to be competitive.

CPLEX ([Cpl93]) is a commercial linear programming system not freely available; it is the object of our future research to compare with it.

## 5 Conclusions

In this paper we have presented TSAT++, an effective system for temporal reasoning that improves the state-of-the-art both on randomly generated, real world and hand-made problems. TSAT++ enforces a number of reasoning techniques and optimizations, both borrowed from the AI and Formal Methods literature and new, and takes full advantage of recent SAT improvements. Thanks to this, TSAT++ outperforms all its competitors, *in each different problem class*. This is particularly remarkable, given that most competitors are optimized or even customized for solving specific classes of problems.

*Acknowledgments* This research is partially supported by MIUR (Italian Ministry of Education, University and Research) under the project RoboCare – A Multi-Agent System with Intelligent Fixed and Mobile Robotic Components. Also, we wish to thank Gilles Audemard, Angelo Oddi, Ofer Strichman and Ioannis Tsamardinos for providing assistance with their solvers.

## References

[ABC+02] Gilles Audemard, Piergiorgio Bertoli, Alessandro Cimatti, Artur Kornilowicz, and Roberto Sebastiani. A SAT based approach for solving formulas over boolean and linear mathematical propositions. In Andrei Voronkov, editor, *Automated Deduction – CADE-18*, volume 2392 of *Lecture Notes in Computer Science*, pages 195–210. Springer-Verlag, July 27-30 2002.

[ACG00] Alessandro Armando, Claudio Castellini, and Enrico Giunchiglia. SAT-based procedures for temporal reasoning. In Susanne Biundo and Maria Fox, editors, *Proceedings of the 5th European Conference on Planning (Durham, UK)*, volume 1809 of *Lecture Notes in Computer Science*, pages 97–108. Springer, 2000.

[ACG+04] Alessandro Armando, Claudio Castellini, Enrico Giunchiglia, Massimo Idini, and Marco Maratea. TSAT++: an open platform for satisfiability modulo theories. Elsevier Science Publishers, 2004. Proceedings of PDPAR 2004 - Pragmatics of Decision Procedures in Automated Reasoning, Cork, Ireland. To appear.

[BM96] R. J. Bayardo, Jr. and D. P. Miranker. A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In *Proc. AAAI*, pages 298–304, 1996.

[BS97] Roberto J. Bayardo, Jr. and Robert C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the 14th National Conference on Artificial Intelligence and 9th Innovative Applications of Artificial Intelligence Conference (AAAI-97/IAAI-97)*, pages 203–208, Menlo Park, July 27–31 1997. AAAI Press.

[BS03] Le Berre and Simon. The essentials of the SAT 2003 competition. In *International Conference on Theory and Applications of Satisfiability Testing (SAT), LNCS*, volume 6, 2003.

[CLR98] Thomas H. Cormen, Charles E. Leiserson, and Ronald R. Rivest. *Introduction to Algorithms*. MIT Press, 1998.

[Cpl93] CPLEX user's guide. Manual, CPLEX Optimization, Inc., Incline Village, NV, USA, 1993.

[DMP91] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49(1-3):61–95, January 1991.

[GC97] Alfonso Gerevini and Matteo Cristani. On finding a solution in temporal constraint satisfaction problems. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 1460–1465, San Francisco, August 23–29 1997. Morgan Kaufmann Publishers.

[GMT03] E. Giunchiglia, M. Maratea, and A. Tacchella. Look-ahead vs. look-back techniques in a modern SAT solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT).*, Portofino, Italy, May 5–8 2003.

[LA97] Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 366–371, San Francisco, August 23–29 1997. Morgan Kaufmann Publishers.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, June 2001.

[OC00] A. Oddi and A. Cesta. Incremental forward checking for the disjunctive temporal problem. In *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI-2000)*, pages 108–112, Berlin, 2000.

[Pro93] Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, 1993.

[SK98] Kostas Stergiou and Manolis Koubarakis. Backtracking algorithms for disjunctions of temporal constraints. In *Proc. AAAI*, 1998.

[SS99] Joao P. Marques Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEETC: IEEE Transactions on Computers*, 48, 1999.

[SSB02] Ofer Strichman, Sanjit A. Seshia, and Randal E. Bryant. Deciding separation formulas with SAT. *Lecture Notes in Computer Science*, 2404:209–222, 2002.

[TP03] Ioannis Tsamardinos and Martha Pollack. Efficient solution techniques for disjunctive temporal reasoning problems. *Artificial Intelligence*, 2003. To appear.

[WW99] Steven Wolfman and Daniel Weld. The LPSAT-engine & its application to resource planning. In *Proc. IJCAI-99*, 1999.