DPLL(T) with Exhaustive Theory Propagation and its Application to Difference Logic

Robert Nieuwenhuis and Albert Oliveras[⋆]

Abstract. At CAV'04 we presented the DPLL(T) approach for satisfiability modulo theories T. It is based on a general DPLL(X) engine whose X can be instantiated with different theory solvers $Solver_T$ for conjunctions of literals.

Here we go one important step further: we require $Solver_T$ to be able to detect *all* input literals that are T-consequences of the partial model that is being explored by DPLL(X). Although at first sight this may seem too expensive, we show that for *difference logic* the benefits compensate by far the costs.

Here we describe and discuss this new version of DPLL(T), the DPLL(X) engine, and our $Solver_T$ for difference logic. The resulting very simple DPLL(T) system importantly outperforms the existing techniques for this logic. Moreover, it has very good scaling properties: especially on the larger problems it gives improvements of orders of magnitude w.r.t. the existing state-of-the-art tools.

1 Introduction

During the last years the performance of decision procedures for the satisfiability of propositional formulas has improved spectacularly. Most state-of-the-art SAT solvers [MMZ$^+$01,GN02] today are based on different variations of the Davis-Putnam-Logemann-Loveland (DPLL) procedure [DP60,DLL62].

But, in many verification applications, satisfiability problems arise for logics that are more expressive than just propositional logic. In particular, decision procedures are required for (specific classes of) ground first-order formulas with respect to theories T such as equality with uninterpreted functions (EUF), the theory of the integer/real numbers, or of arrays or lists.

Normally, for *conjunctions* of theory literals there exist well-studied decision procedures. For example, such a *theory solver* for the case where T is equality (i.e., for EUF logic) can use *congruence closure*. It runs in $O(n \log n)$ time [DST80], also in the presence of successor and predecessor functions [NO03]. Another example is *difference logic* (sometimes also called *separation logic*) over the integers or reals, where atoms take the form $a - b \leq k$, being a and b variables and k a constant. In difference logic the satisfiability of conjunctions of such literals can be decided in $O(n^3)$ time by the Bellman-Ford algorithm.

[⋆] Technical Univ. of Catalonia, Barcelona, `www.lsi.upc.es/~roberto|~oliveras`. Partially supported by Spanish Min. of Educ. and Science through the LogicTools project (TIN2004-03382, both authors), and FPU grant AP2002-3533 (Oliveras).

However, it is unclear in general which is the best way to handle arbitrary Boolean or CNF formulas over theory (and propositional) literals. Typically, the problem is attacked by trying to combine the strengths of the DPLL approach for dealing with the boolean structure, with the strengths of the specialized procedures for handling conjunctions of theory literals.

One well-known possibility is the so-called *lazy approach* [ACG00,dMR02] [BDS02,FJOS03,BBA$^+$05]. In this approach, each theory atom is simply abstracted by a distinct propositional variable, and a SAT solver is used to find a propositional model. This model is then checked against the theory by using the solver for conjunctions of literals. Theory-inconsistent models are discarded from later consideration by adding an appropriate *theory lemma* to the original formula and restarting the process. This is repeated until a model compatible with the theory is found or all possible propositional models have been explored. The main advantage of such lazy approaches is their flexibility: they can easily combine new decision procedures for different logics with new SAT solvers. Nowadays, most lazy approaches have tighter integrations in which partial propositional models are checked incrementally against the theory while they are built by the SAT solver. This increases efficiency at the expense of flexibility.

However, these lazy approaches suffer from the drawbacks of *insufficient constraint propagation* and *blind search* [dMR04]: essentially, the theory information is used only to *validate* the search a posteriori, not to *guide* it a priori.

In practice, for some theories these lazy approaches are outperformed by the so-called *eager* techniques, where the input formula is translated, in a single satisfiability-preserving step, into a propositional CNF, which is checked by a SAT solver for satisfiability. However, such eager approaches require sophisticated ad-hoc translations for each logic. For example, for EUF there exist the *per-constraint* encoding [BV02], the *small domain* encoding [PRSS99,BLS02], and several hybrid approaches [SLB03]. Similarly, for difference logic, sophisticated range-allocation approaches have been defined in order to improve the translations [TSSP04]. But, in spite of this, on many practical problems the translation process or the SAT solver run out of time or memory (see [dMR04]).

1.1 The DPLL(T) approach of [GHN$^+$04]

As a way to overcome the drawbacks of the lazy and eager approaches, at CAV'04 we proposed DPLL(T) [GHN$^+$04]. It consists of a general DPLL(X) engine, whose parameter X can be instantiated with a solver (for conjunctions of literals) $Solver_T$ for a given theory T, thus producing a DPLL(T) decision procedure.

One essential aspect of DPLL(T) is that $Solver_T$ not only validates the choices made by the SAT engine (as in the lazy approaches). It also eagerly detects literals of the input CNF that are T-consequences of the current partial model, and sends them to the DPLL(X) engine for propagation. Due to this, for the EUF logic the DPLL(T) approach not only outperforms the lazy approaches, but also all eager ones, as soon as equality starts playing a significant role in the

EUF formula [GHN$^+$04][1]. On the other hand, DPLL(T) is similar in flexibility to the lazy approaches: other logics can be dealt with by simply plugging in their theory solvers into the DPLL(X) engine, provided that these solvers conform to a minimal and simple interface.

In the DPLL(T) version of [GHN$^+$04], Solver_T is allowed to fail sometimes to detect that a certain input literal l is a T-consequence of literals $l_1, \ldots, l_n$ in the current partial model. Then, only when the DPLL(X) engine actually makes $\neg l$ true, as a decision literal, or as a unit propagated literal, and communicates this to Solver_T, it is detected that the partial model is no longer T-consistent. Then Solver_T warns the DPLL(X) engine, who has several different ways of treating such situations. One possibility is to backjump to the level where l actually became T-entailed, and propagate it there. This mechanism alone gives one a complete DPLL(T) decision procedure. But in order to make it more efficient, it is usually better to *learn* the corresponding theory lemma $l_1 \wedge \ldots \wedge l_n \rightarrow l$. In other similar branches of the DPLL search the literal l can then be propagated earlier. Altogether, such concrete situations of non-exhaustiveness of Solver_T are essentially handled as in the lazy approaches.

The reason why in [GHN$^+$04] the approach was defined considering a possibly non-exhaustive Solver_T was due to our experiments with EUF. More precisely, for *negative* equality consequences we found it expensive to detect them exhaustively, whereas all positive literals were propagated.

1.2 DPLL(T) with Exhaustive Theory Propagation

At least for difference logic, it is indeed possible to go one important step further in this idea: in this paper we describe a DPLL(T) approach where Solver_T is required to detect and communicate to DPLL(X) *all* literals of the input formula that are T-consequences of the partial model that is being explored. This assumption makes the DPLL(X) engine much simpler and efficient than before, because it can propagate these literals in exactly the same way as for standard unit propagation in DPLL, and no theory lemma learning is required at all.

The DPLL(X) engine then becomes essentially a propositional SAT solver. The only difference is a small interface with Solver_T. DPLL(X) communicates to Solver_T each time the truth value of a literal is set, and Solver_T answers with the list of literals that are new T-consequences. DPLL(X) also communicates to Solver_T, each time a backjump takes place, how many literals of the partial interpretation have been unassigned. As in most modern DPLL systems, backjumping is guided by an *implication graph* [MSS99], but of course here some arrows in the graph correspond to theory consequences. Therefore, for building the graph, DPLL(X) also needs Solver_T to provide an $\mathit{Explain}(l)$ operation, returning, for each T-consequence l it has communicated to DPLL(X), a (preferably small) subset of the true literals that implied l. This latter requirement to our Solver_T coincides with what solvers in the lazy approach must do for returning the theory lemmas.

[1] And our implementations are now again much faster than reported at CAV'04.

Due to the fact that DPLL(X) is here nothing more than a standard DPLL-based SAT solver, the approach has become again more flexible, because it is now easy to convert any new DPLL-based SAT solver into a DPLL(X) engine. Moreover, there is at least one important theory for which the exhaustiveness requirement does not make $Solver_T$ too slow: here we give extensive experimental evidence showing that for difference logic our approach outperforms all existing systems, and moreover has better scaling properties. Especially on the larger problems it gives improvements of orders of magnitude.

This paper is structured as follows. In Section 2 we give a precise formulation of DPLL(T) with exhaustive theory propagation. In Section 3 we show how our relatively simple solver for difference logic is designed in order to efficiently fulfill the requirements. Section 4 gives all experimental results, and we conclude in Section 5.

2 DPLL(T): basic definitions and notations

A procedure for Satisfiability Modulo Theories (SMT) is a procedure for deciding the satisfiability of ground (in this case, CNF) formulas in the context of a background theory T. By *ground* we mean containing no variables—although possibly containing constants not in T (which can also be seen as Skolemized existential variables).

A *theory* T is a satisfiable set of closed first-order formulas. We deal with (partial Herbrand) interpretations M as sets of ground literals such that $\{A, \neg A\} \subseteq M$ for no ground atom A. A ground literal l is *true* in M if $l \in M$, is *false* in M if $\neg l \in M$, and is *undefined* otherwise. A ground clause C is true in M if $C \cap M \neq \emptyset$. It is false in M, denoted $M \models \neg C$, if all its literals are false in M. Similarly, we define in the standard way when M satisfies (is a model of) a theory T. If F and G are ground formulas, *G is a T-consequence of F* written $F \models_T G$, if $T \cup F \models G$. The decision problem that concerns us here is whether a ground formula F is *satisfiable in* a theory T, that is, whether there is a model M of $T \cup F$. Then we say that M is a T-model of F.

2.1 Abstract transition rules

Here we define DPLL(T) with exhaustive theory propagation by means of the *abstract DPLL* framework, introduced in [NOT05] (check this reference for details). Here a DPLL procedure is modelled by a transition relation over states. A state is either *fail* or a pair $M \parallel F$, where F is a finite set of clauses and M is a sequence of literals that is seen as a partial interpretation. Some literals l in M will be *annotated* as being *decision literals*; these are the ones added to M by the Decide rule given below, and are sometimes written l^{d}. The transition relation is defined by means of rules.

Definition 1. *The* DPLL system with exhaustive theory propagation *consists of the following transition rules:*

UnitPropagate :

$$M \parallel F, C \vee l \implies M\, l \parallel F, C \vee l \quad \textbf{if} \begin{cases} M \models \neg C \\ l \textit{ is undefined in } M \end{cases}$$

Decide :

$$M \parallel F \implies M\, l^{\mathsf{d}} \parallel F \quad \textbf{if} \begin{cases} l \textit{ or } \neg l \textit{ occurs in a clause of } F \\ l \textit{ is undefined in } M \end{cases}$$

Fail :

$$M \parallel F, C \implies \textit{fail} \quad \textbf{if} \begin{cases} M \models \neg C \\ M \textit{ contains no decision literals} \end{cases}$$

Backjump :

$$M\, l^{\mathsf{d}}\, N \parallel F, C \implies M\, l' \parallel F, C \quad \textbf{if} \begin{cases} M\, l^{\mathsf{d}}\, N \models \neg C, \textit{ and there is} \\ \textit{some clause } C' \vee l' \textit{ s.t.:} \\ \quad F, C \models_T C' \vee l' \textit{ and } M \models \neg C' \\ \quad l' \textit{ is undefined in } M \\ \quad l' \textit{ or } \neg l' \textit{ occurs in } F \textit{ or } M\, l^{\mathsf{d}}\, N \end{cases}$$

T-Propagate :

$$M \parallel F \implies M\, l \parallel F \quad \textbf{if} \begin{cases} M \models_T l \\ l \textit{ or } \neg l \textit{ occurs in a clause of } F \\ l \textit{ is undefined in } M \end{cases}$$

Learn :

$$M \parallel F \implies M \parallel F, C \quad \textbf{if} \begin{cases} \textit{all atoms of } C \textit{ occur in } F \\ F \models_T C \end{cases}$$

Forget :

$$M \parallel F, C \implies M \parallel F \quad \textbf{if} \ \{ F \models_T C$$

Restart :

$$M \parallel F \implies \emptyset \parallel F$$

These rules express how the search state of a DPLL procedure evolves. Without T-Propagate, and replacing everywhere $\models_T$ by $\models$, they model a standard propositional DPLL procedure. Note that this is equivalent to considering T to be the empty theory: then T-Propagate never applies. The propagation and decision rules extend the current partial interpretation M with new literals, and if in some state $M \parallel F$ there is a *conflict*, i.e., a clause of F that is false in M, always either Fail applies (if there are no decision literals in M) or Backjump applies (if there is at least one decision literal in M). In the latter case, the *backjump clause* $C' \vee l'$ can be found efficiently by constructing a *conflict graph.* Good backjump clauses allow one to return to low *decision levels*, i.e., they maximize the number of literals in N. Usually such backjump clauses are learned by the Learn rule, in order to prevent future similar conflicts. The use of Forget is to free memory by removing learned clauses that have become less active (e.g., cause less conflicts or propagations).

These first five rules terminate (independently of any strategies or priorities) and termination of the full system is easily enforced by limiting the applicability of the other three rules (e.g., if all Learn steps are immediately after Backjump steps, and Restart is done with increasing periodicity). If we say that a state is *final* if none of the first five rules applies, the following theorem is proved in a similar way to what is done in [NOT05].

Theorem 2. *Let $\Longrightarrow$ denote the transition relation of the DPLL system with exhaustive theory propagation where T-Propagate is applied eagerly, i.e., no other rule is applied if T-Propagate is applicable, and let $\Longrightarrow^*$ be its transitive closure.*

1. *$\emptyset \parallel F \Longrightarrow^*$ fail if, and only if, F is unsatisfiable in T.*
2. *If $\emptyset \parallel F \Longrightarrow^* M \parallel F'$, where $M \parallel F'$ is final, then M is a T-model of F.*

2.2 Our particular strategy

Of course, actual DPLL implementations may use the above rules in more restrictive ways, using particular application strategies. For example, many systems will eagerly apply UnitPropagate and minimize the application of Decide, but this is not necessary: any strategy will be adequate for Theorem 2 to hold.

We now briefly explain the particular strategy used by our DPLL(T) implementation, and the roles of the DPLL(X) engine and of $Solver_T$ in it.

For the initial setup of DPLL(T), one can consider that it is $Solver_T$ that reads the input CNF, then stores the list of all literals occurring in it, and hands it over to DPLL(X) as a purely propositional CNF. After that, DPLL(T) implements the rules as follows:

- Each time DPLL(X) communicates to $Solver_T$ that the truth value of a literal has been set, due to UnitPropagate or Decide, $Solver_T$ answers with the list of all input literals that are new T-consequences. Then, for each one of these consequences, T-Propagate is immediately applied.
- If T-Propagate is not applicable, then UnitPropagate is eagerly applied by DPLL(X) (this is implemented using the two-watched-literals scheme).
- DPLL(X) applies Fail or Backjump if, and only if, a conflict clause C is detected, i.e., a clause C that is false in M. As said, if there is some decision literal in M, then it is always Backjump that is applied. The application of Backjump is guided by an *implication graph.* Each literal of the conflict clause C is false in M because its negation l is in M, which can be due to one of three possible rules:
  - UnitPropagate: l is true because, in some clause $D \vee l$, every literal in D is the negation of some l' in M.

  - T-Propagate: l has become true because it is a T-consequence of other literals l' in M.
  - Decide: l has been set true by Decide.

  In the cases of UnitPropagate and T-Propagate, recursively the l' are again true due to the same three possible reasons. By working backwards in this way from the literals of C, one can trace back the reasons of the conflict. A conflict graph is nothing but a representation of these reasons. By analyzing a subset of it, one can find adequate backjump clauses [MSS99]. But for building the graph, for the case of the T-Propagate implications, $Solver_T$ must be able to return the set of l' that T-entailed l. This is done by the $Explain(l)$ operation provided by $Solver_T$.
  After each backjump has taken place in DPLL(X), it tells $Solver_T$ how many literals of the partial interpretation have been unassigned.
- Immediately after each Backjump application, the Learn rule is applied for learning the backjump clause.
- In our current implementation, DPLL(X) applies Restart when certain system parameters reach some limits, such as the number of conflicts or lemmas, the number of new units derived, etc.
- Forget is applied by DPLL(X) after each restart (and only then), removing at least half of the lemmas according to their activity (number of times involved in a conflict since last restart). The 500 newest lemmas are not removed.
- DPLL(X) applies Decide only if none of the other first five rules is applicable. The heuristic for chosing the decision literal is as in Berkmin [GN02]: we take an unassigned literal that occurs in an as recent as possible lemma, and in case of a draw, or if there is no such literal in the last 100 lemmas, the literal with the highest VSIDS measure is taken [MMZ$^+$01] (where each literal has a counter increased by each participation in a conflict, and from time to time all counters are divided by a constant).

3 Design of $Solver_T$ for Difference Logic

In this section we address the problem of designing $Solver_T$ for a DPLL(T) system deciding the satisfiability of a CNF formula F in difference logic (sometimes also called separation logic). In this logic, the domain can be the integers, rationals or reals (as we will see, the problem is essentially equivalent in all three cases), and atoms are of the form $a \leq b+k$, where a and b are variables over this domain and k is a constant.

Note that, over the integers, atoms of the form $a < b+k$ can be equivalently written as $a \leq b + (k-1)$. A similar transformation exists for rationals and reals, by decreasing k by a small enough amount that depends only on the remaining literals ocurring in the input formula [Sch87]. Hence, negations can also be removed, since $\neg(a \leq b+k)$ is equivalent to $b < a-k$, as well as equalities $a = b+k$, which are equivalent to $a \leq b+k \ \wedge\ a \geq b+k$. Therefore, we will consider that all literals are of the form $a \leq b+k$.

Given a conjunction of such literals, one can build a directed weighted graph whose nodes are the variables, and with an edge $a \xrightarrow{k} b$ for each literal $a \leq b+k$. It is easy to see that, independently of the concrete arithmetic domain (i.e., integers, rationals or reals), such a conjunction is unsatisfiable if, and only if, there is a cycle in the graph with negative accumulated weight. Therefore, once the problem has all its literals of the form $a \leq b + k$, the concrete domain does not matter any more.

Despite its simplicity, difference logic has been used to express important practical problems, such as verification of timed systems, scheduling problems or the existence of paths in digital circuits with bounded delays.

3.1 Initial setup

As said, for the initial setup of DPLL(T), it is *Solver*$_T$ that reads the input CNF, then stores the list of all literals occurring in it, and hands it over to DPLL(X) as a purely propositional CNF.

For efficiency reasons, it is important that in this CNF the relation between literals and their negations is made explicit. For example, if $a \leq b + 2$ and $b \leq a - 3$ occur in the input, then, since (in the integers) one is the negation of the other, they should be abstracted by a propositional variable and its negation. This can be detected by using a canonical form during this setup process. For instance, one can impose that always the smallest variable, say a, has to be at the left-hand side of the $\leq$ relation, and thus we would have $a \leq b + 2$ and $\neg(a \leq b + 2)$, and abstract them by p and $\neg p$ for some propositional variable p.

Solver$_T$ will keep a data structure recording all such canonized input literals like $a \leq b + 2$ and its abstraction variable p. Moreover, for reasons we will see below, it keeps for each variable the list of all input literals it occurs in, together with the length of this list.

3.2 DPLL(X) sets the truth value of a literal

When the truth value of a literal is set, *Solver*$_T$ converts the literal into the form $a \leq b + k$ and adds the corresponding edge to the aforementioned directed weighted graph. Since there is a one-to-one correspondence between edges and such literals, and between the graph and the conjunction of the literals, we will sometimes speak about literals that are (T-)consequences of the graph. Here we will write $a_0 \xrightarrow{k\ *} a_n$ if there is a path in the graph of the form

$$a_0 \xrightarrow{k_1} a_1 \xrightarrow{k_2} \ldots \xrightarrow{k_{n-1}} a_{n-1} \xrightarrow{k_n} a_n$$

with $n \geq 0$ and where $k = 0 + k_1 + \ldots k_n$ is called the length of this path.

Note that one can assume that DPLL(X) does not communicate to *Solver*$_T$ any redundant edges, since such consequences would already have been communicated by *Solver*$_T$ to DPLL(X). Similarly, DPLL(X) will not communicate to *Solver*$_T$ any edges that are inconsistent with the graph. Therefore, there will be no cycles of negative length.

Here, $Solver_T$ must return to DPLL(X) all input literals that are new consequences of the graph once the new edge has been added. Essentially, for detecting the new consequences of a new edge $a \xrightarrow{k} b$, $Solver_T$ needs to check all paths

$$a_i \xrightarrow{k_i\ *} a \xrightarrow{k} b \xrightarrow{k'_j\ *} b_j$$

and see whether there is any input literal that follows from $a_i \leq b_j + (k_i + k + k'_j)$, i.e., an input literal of the form $a_i \leq b_j + k'$, with $k' \geq k_i + k + k'_j$.

For checking all such paths from a_i to b_j that pass through the new edge from a to b, we need to be able to find all nodes a_i from which a is reachable, as well as the nodes b_j that are reachable from b. Therefore, we keep the graph in double adjacency list representation: for each node n, we keep the list of outgoing edges as well as the one of incoming edges. Then a standard single-source-shortest-path algorithm starting from a can be used for computing all a_i with their corresponding minimal k_i (and similarly for the b_j).

What worked best in our experience to finally detect all entailed literals is the following. We use a simple depth-first search, where each time a node is reached for the first time it is marked, together with the accumulated distance k, and, each time it is reached again with some k', the search stops if $k' \geq k$ (this terminates because there are no cycles of negative length).

While doing this, the visited nodes are pushed onto two stacks, one for the a_i's and another one for the b_j's, and it is also counted, for each one of those two stacks, how many input literals these a_i's (and b_j's) occur in (remember that there are precomputed lists for this, together with their lengths).

Then, if, w.l.o.g., the a_i's are the ones that occur in less input literals, we check, for each element in the list of input literals containing each a_i, whether the other constant is some of the found b_j, and whether the literal is entailed or not (this can be checked in constant time since previously all b_j have been marked).

3.3 Implementation of Explain

As said, for building the implication graph, DPLL(X) needs $Solver_T$ to provide an *Explain*(l) operation, returning, for each T-consequence l it has communicated to DPLL(X), a preferably small subset of the literals that implied l.

For implementing this, we proceed as follows. Whenever the m-th edge is added to the directed weighted graph, the edge is annotated with its associated insertion number m. In a similar fashion, when a literal l is returned as a consequence of the m-th edge, this m is recorded together with l. Now assume l is of the form $a \leq b + k$, and the explanation for l is required. Then we search a path in the graph from a to b of length at most k, using a depth-first search as before. Moreover, in this search we will not traverse any edges with insertion number greater than m. This not only improves efficiency, but it is it is also needed for not returning "too new" explanations, which may create cycles in the implication graph, see [GHN$^+$04].

4 Experimental Evaluation

Experiments have been done with all benchmarks we know of for difference logic, both real-world and hand-made ones[2]. The table below contains runtimes for five suites of benchmark families: the SAL Suite [dMR04], the MathSAT Suite (see `mathsat.itc.it`), and the DLSAT one [CAMN04] come from verification by bounded model checking of timed automated and linear hybrid systems and from the job shop scheduling problem (the `abz` family of DLSAT). The remaining two suites are hand-made. The Diamond Suite is from the problem generator of [SSB02], where problem `diamondsN` has N edges per diamond, generating between 10 and 18 diamonds (i.e., 9 problems per family), forcing unsatisfiability over the integers. The DTP Suite is from [ACGM04].

We compare with three other systems: ICS 2.0 (`ics.csl.sri.com`), MathSAT [BBA$^+$05][3] and TSAT++ (see [ACGM04] and `ai.dist.unige.it/Tsat`, we thank Claudio Castellini for his help with this system). For the handmade problems, TSAT++ has been used in the setting recommended by the authors for these problems; for the other problems, we used the best setting we could find (as recommended to us by the authors). DPLL(T) has been used on all problems in the same standard setting, as described in this paper.

We have included ICS and not others such as CVC [BDS02], CVC-Lite [BB04], UCLID [LS04], because, according to [dMR04], ICS either dominates them or gives similar results. It has to be noted that UCLID could perhaps improve its performance by using the most recent range allocation techniques of [TSSP04], and that ICS applies a more general solver for linear arithmetic, rather than a specialized solver for difference logic as MathSAT, TSAT++ and DPLL(T) do.

On all benchmark families, DPLL(T) is always significantly better than all other systems. It is even orders of magnitude faster, especially on the larger problems, as soon as the theory becomes relevant, i.e., when in, say, at least 10 percent of the conflicts the theory properties play any role. This is the case for all problem families except lpsat and the FISCHER problems of the MathSAT Suite.

Results are in seconds and are aggregated per family of benchmarks, with times greater than 100s rounded to whole numbers. All experiments were run on a 2GHz 512MB Pentium-IV under Linux. Each benchmark was run for one hour, i.e., 3600 seconds. An annotation of the form (n t) or (n m) in a column indicates respectively that the system timed out or ran out of memory on n benchmarks. Each timeout or memory out is counted as 3600s.

[2] Individual results for each benchmark can be found at `www.lsi.upc.es/~oliveras`, together with all the benchmarks and an executable of our system.

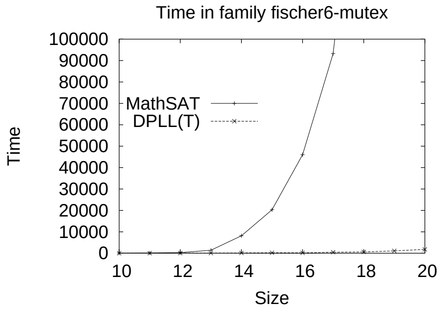
[3] V3.1.0, Nov 22, 2004, see `mathsat.itc.it`, which features a new specialized solver for difference logic. We have no exhaustive results yet of the even more recent V3.1.1 of Jan 12, 2005 on all the larger problems. It appears to be slightly faster than V3.1.0 on some problems, but with results relative to DPLL(T) similar to V3.1.0. We will keep up-to-date results on `www.lsi.upc.es/~oliveras`.

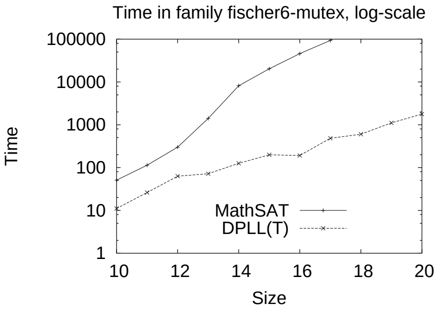
Benchmark family	# Problems in family	ICS	MathSAT	TSAT++	DPLL(T)
SAL Suite:					
lpsat	20	636	185	490	135
bakery-mutex	20	39.44	17.91	9.93	0.5
fischer3-mutex	20	(7t) 27720	363	(2t) 14252	259
fischer6-mutex	20	(10t) 39700	(7t) 27105	(11t) 40705	4665
fischer9-mutex	20	(12t) 43269	(9t) 33380	(13t) 48631	(2t) 14408
MathSAT Suite:					
FISCHER9	10	187	187	172	86.68
FISCHER10	11	1162	962	3334	380
FISCHER11	12	(1t) 5643	4037	(2t) 9981	3091
FISCHER12	13	(3t) 11100	(2t) 8357	(4t) 14637	(1t) 6479
FISCHER13	14	(4t) 14932	(3t) 12301	(5t) 18320	(2t) 10073
FISCHER14	15	(5t) 18710	(4t) 15717	(6t) 218891	(3t) 14253
PO4	11	14.57	33.98	28.01	2.68
PO5	13	(10m) 36004	269	220	23.8
DLSAT Suite:					
abz	12	(2t) 11901	218	49.02	5.29
ba-max	19	770	211	233	14.55
Diamond Suite:					
diamonds4	9	(2m) 11869	9018	501	312
diamonds6	9	(2m) 9054	2926	742	193
diamonds10	9	(2m) 11574	(1t) 4249	1567	207
diamonds20	9	(4m, 1t) 19286	5050	(1t) 6073	219
DTP Suite:					
DTP-175	20	(8t) 45060	37.63	35.69	0.77
DTP-210	20	(20t) 72000	50.74	112	5.27
DTP-240	20	(20t) 72000	36.53	191	6.86

4.1 Scaling Properties

To illustrate the scaling properties of our approach, below we include two graphical representations of the behaviour of MathSAT and DPLL(T) on the fischer6-mutex family, which is a typical real-world suite for which large benchmarks exist where the theory plays a relevant role (the other such suites give similar graphics).

The diagram on the left below compares both systems on the problems of size between 10 and 20 on a normal time scale of up to 100,000 seconds. MathSAT did not finish any of the problems 18, 19 and 20 in 210,000 seconds, whereas DPLL(T) (almost invisible) takes 603, 1108 and 1778 seconds on them, respectively. The diagram on the right expresses the same results on a logarithmic scale, in order to get a better impression of the asymptotic behaviour of DPLL(T).





5 Conclusions and Further Work

We have shown that it is possible to deal with Satisfiability Modulo Theories (SMT) in a clean and modular way, even if the information for the theory under consideration is used exhaustively for propagating implied literals. Although at first sight one might get the impression that this may be too expensive, we have shown that, at least for difference logic, this is not the case.

Future work concerns other theories for which exhaustive theory propagation may be useful, and others where a hybrid approach has to be applied, i.e., where some classes of unit T-consequences are assumed to be detected and other are handled more lazily.

References

[ACG00] A. Armando, C. Castellini, and E. Giunchiglia. SAT-based procedures for temporal reasoning. In *Procs 5th European Conf. on Planning (Durham, UK)*, LNCS 1809 pages 97–108. Springer, 2000.

[ACGM04] A. Armando, C. Castellini, E. Giunchiglia, and M. Maratea. A SAT-based Decision Procedure for the Boolean Combination of Difference Constraints. In *7th Int. Conf. Theory and Appl. of Sat Testing(SAT 2004)*. LNCS, 2004.

[BB04] C. Barrett and S. Berezin. CVC lite: A new implementation of the cooperating validity checker. In *16th Int. Conf. Computer Aided Verification, CAV'04* LNCS 3114, pages 515–518. Springer, 2004.

[BBA$^+$05] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P .v. Rossum, S. Schulz, and R. Sebastiani. An incremental and layered procedure for the satisfiability of linear arithmetic logic. In *TACAS'05*, 2005.

[BDS02] C. Barrett, D. Dill, and A. Stump. Checking satisfiability of first-order formulas by incremental translation into sat. In *Procs. 14th Intl. Conf. on Computer Aided Verification (CAV)*, LNCS 2404, 2002.

[BLS02] R. Bryant, S. Lahiri, and S. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Procs. 14th Intl. Conference on Computer Aided Verification (CAV)*, LNCS 2404, 2002.

[BV02] R. Bryant and M. Velev. Boolean satisfiability with transitivity constraints. *ACM Trans. Computational Logic*, 3(4):604–627, 2002.

[CAMN04] S. Cotton, E. Asarin, O. Maler, and P. Niebert. Some progress in satisfiability checking for difference logic. In *FORMATS 2004 and FTRTFT 2004*, LNCS 3253, pages 263–276, 2004.

[DLL62] M. Davis, G. Logemann, and D .Loveland. A machine program for theorem-proving. *Comm. of the ACM*, 5(7):394–397, 1962.

[dMR02] L. de Moura and H. Rueß. Lemmas on demand for satisfiability solvers. In *Procs. 5th Int. Symp. on the Theory and Applications of Satisfiability Testing, SAT'02*, pages 244–251, 2002.

[dMR04] L. de Moura and H. Ruess. An experimental evaluation of ground decision procedures. In *16th Int. Conf. on Computer Aided Verification, CAV'04*, LNCS 3114, pages 162–174. Springer, 2004.

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[DST80] P. Downey, R. Sethi, and R. Tarjan. Variations on the common subexpressions problem. *Journal of the ACM*, 27(4):758–771, 1980.

[FJOS03] C. Flanagan, R. Joshi, X. Ou, and J. Saxe. Theorem proving using lazy proof explanation. In *Procs. 15th Int. Conf. on Computer Aided Verification (CAV)*, LNCS 2725, 2003.

[GHN$^+$04] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In *16th Int. Conf. on Computer Aided Verification, CAV'04*, LNCS 3114, pp 175–188, 2004.

[GN02] E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT-solver. In *Design, Automation, and Test in Europe (DATE '02)*, pages 142–149, 2002.

[LS04] S. Lahiri and S. Seshia. The UCLID Decision Procedure. In *Computer Aided Verification, 16th International Conference, (CAV'04)*, Lecture Notes in Computer Science, pages 475–478. Springer, 2004.

[MMZ$^+$01] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proc. 38th Design Automation Conference (DAC'01)*, 2001.

[MSS99] J. Marques-Silva and K. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Comput.*, 48(5):506–521, 1999.

[NO03] Robert Nieuwenhuis and Albert Oliveras. Congruence closure with integer offsets. In M Vardi and A Voronkov, eds, *10h Int. Conf. Logic for Programming, Artif. Intell. and Reasoning (LPAR)*, LNAI 2850, pp 78–90, 2003.

[NOT05] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Abstract DPLL and Abstract DPLL Modulo Theories. In *11h Int. Conf. Logic for Programming, Artif. Intell. and Reasoning (LPAR)*, LNAI 3452, 2005.

[PRSS99] A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel. Deciding equality formulas by small domains instantiations. In *Procs. 11th Int. Conf. on Computer Aided Verification (CAV)*, LNCS 1633, pages 455–469, 1999.

[Sch87] A. Schrijver. *Theory of Linear and Integer Programming.* John Wiley and Sons, New York, 1987.

[SLB03] S. Seshia, S. Lahiri, and R. Bryant. A hybrid SAT-based decision procedure for separation logic with uninterpreted functions. In *Procs. 40th Design Automation Conference (DAC)*, pages 425–430, 2003.

[SSB02] O. Strichman, S. Seshia, and R. Bryant. Deciding separation formulas with SAT. In *Procs. 14th Intl. Conference on Computer Aided Verification (CAV)*, LNCS 2404, pages 209–222, 2002.

[TSSP04] M. Talupur, N. Sinha, O. Strichman, and A. Pnueli. Range allocation for separation logic. In *16th Int. Conf. on Computer Aided Verification, CAV'04*, LNCS 3114, pp 148–161, 2004.