Decision procedures for SAT, SAT Modulo Theories and Beyond. The BarcelogicTools*

Robert Nieuwenhuis and Albert Oliveras**

Abstract. An overview is given of a number of recent developments in SAT and SAT Modulo Theories (SMT). In particular, based on our framework of Abstract DPLL and Abstract DPLL modulo Theories, we explain our DPLL(T) approach to SMT.
Experimental results and future projects are discussed within BarcelogicTools, a set of logic-based tools developed by our research group in Barcelona. At the 2005 SMT competition, BarcelogicTools won all four categories it participated in (out of the seven existing categories).

1 Introduction

Nowadays, SAT solvers and their extensions are becoming the tool of choice for attacking more and more different problems in areas such as Electronic Design Automation, Verification, Artificial Intelligence, or Operations Research. Most state-of-the-art SAT solvers [MMZ$^+$01,GN02,ES03,Rya04] today are based on the Davis-Putnam-Logemann-Loveland (DPLL) procedure [DP60,DLL62]. These DPLL-based SAT solvers have spectacularly improved in the last years, due to better implementation techniques and conceptual enhancements such as *backjumping*, *conflict-driven lemma learning* ([MSS99]), and *restarts*. These advances make it possible to decide the satisfiability of industrial SAT problems with tens of thousands of variables and millions of clauses.

Because of their success, both the DPLL procedure and its enhancements have been adapted for handling satisfiability problems in logics that are more expressive than propositional logic. For example, some properties of timed automata are naturally expressed in *difference logic*, where formulas contain atoms of the form $a - b \leq k$, which are interpreted with respect to a background theory T of the integers, rationals or reals [Alu99]. Similarly, for the verification of pipelined microprocessors it is convenient to consider a logic of *Equality with Uninterpreted Functions (EUF)*, where the background theory T specifies a congruence [BD94]. To mention just one other example, the conditions arising from

---

* Parts of this work were published in a preliminary form in [NO03,GHN$^+$04,NOT05] [NO05c,NO05b,NO05a], several of them in collaboration with Cesare Tinelli.

** 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), FPU grant AP2002-3533 (Oliveras), and by Cesare Tinelli's NSF Career grant #0237422 supporting Oliveras' stays in Iowa.

program verification usually involve arrays, lists and other data structures, so it becomes very natural to consider satisfiability problems *modulo* the theory T of these data structures. In such applications, problems may contain thousands of clauses like

$$p \quad \vee \quad \neg q \quad \vee \quad a = f(b-c) \quad \vee \quad read(s,\, f(b-c)\,) = d \quad \vee \quad a - g(c) \, \leq 7$$

containing purely propositional atoms as well as atoms over (combined) theories. This is known as the *Satisfiability Modulo Theories* (SMT) problem for a theory T: given a formula F, determine whether F is T-satisfiable, i.e., whether there exists a model of T that is also a model of F. A library of benchmarks for SMT called *SMT-LIB* is maintained at `http://combination.cs.uiowa.edu/smtlib/` and a formal standard for its syntax exists [RT03].

In this paper, based on our framework of Abstract DPLL (Section 2) and Abstract DPLL modulo Theories (Section 3), we explain our DPLL(T) approach to SMT (Section 4). We describe two variants of DPLL(T), depending on whether *theory propagation* is done exhaustively or not. DPLL(T) is based on a general DPLL(X) engine, whose parameter X can be instantiated with specialized solvers $Solver_T$ for given theories T, thus producing a system DPLL(T). Once the DPLL(X) engine has been implemented, this approach becomes extremely flexible: new theories can be dealt with by simply plugging in new theory solvers. These solvers must only be able to deal with *conjunctions* of theory literals and conform to a minimal and simple set of additional requirements. We describe how DPLL(X) and $Solver_T$ cooperate, and the architecture of DPLL(T) for several theories that are widely used in industrial verification problems.

Section 5 describes BarcelogicTools, a set of logic-based tools developed by our research group in Barcelona, including, in particular, a state-of-the-art SAT solver, a DPLL(X) engine, and a number of theory solvers. Results show that our DPLL(T) systems in BarcelogicTools significantly outperform the other state-of-the-art tools, frequently in several orders of magnitude, and moreover scale up very well. In fact, at the 2005 SMT competition, BarcelogicTools won all four categories it participated in (out of seven categories that existed in total; search SMT Competition on the web). Finally, some future extensions of the BarcelogicTools project are discussed.

2 Abstract DPLL in the propositional case

Let P be a fixed finite set of propositional symbols. If $p \in P$, then p and $\neg p$ are *literals* of P. The *negation* of a literal l, written $\neg l$, denotes $\neg p$ if l is p, and p if l is $\neg p$. A *clause* is a disjunction of literals $l_1 \vee \ldots \vee l_n$. A *unit clause* is a clause consisting of a single literal. A (finite, non-empty, CNF) *formula* is a conjunction of one or more clauses $C_1 \wedge \ldots \wedge C_n$. When it leads to no ambiguities, we sometimes also write such a formula in set notation $\{C_1, \ldots, C_n\}$ or simply replace $\wedge$ connectives by commas.

A (partial truth) *assignment* M is a set of literals such that $\{p, \neg p\} \subseteq M$ for no p. A literal l is *true* in M if $l \in M$, it is *false* in M if $\neg l \in M$, and l is *undefined* in M otherwise. M is *total* over P if no literal of P is undefined in M. A clause C is true in M if at least one of its literals is in M. It is false in M if all its literals are false in M, and it is undefined in M otherwise. A formula F is true in M, or *satisfied* by M, denoted $M \models F$, if all its clauses are true in M. In that case, M is called a *model* of F. If F has no models then it is called *unsatisfiable*. If F and F' are formulas, we write $F \models F'$ if F' is true in all models of F. Then we say that F' *is entailed by* F, or is a *logical consequence* of F. If $F \models F'$ and $F' \models F$, we say that F and F' are *logically equivalent*.

In what follows, (possibly subscripted or primed) lowercase l *always* denote literals. Similarly C and D always denote clauses, F and G denote formulas, and M and N are assignments. If C is a clause $l_1 \vee \ldots \vee l_n$, we sometimes write $\neg C$ to denote the formula $\neg l_1 \wedge \ldots \wedge \neg l_n$.

2.1 The Classical DPLL Procedure

Here a DPLL procedure is modelled by a transition relation over states (check [NOT05] for details). A state is either *FailState* 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. The following simple Classical DPLL system is given here mainly for explanatory and historical reasons.

Definition 1. *The* Classical DPLL system Cl *consists of the five rules:*

UnitPropagate :

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

PureLiteral :

$$M \parallel F \implies M\, l \parallel F \quad \textbf{if} \begin{cases} l \textit{ occurs in some clause of } F \\ \neg l \textit{ occurs in no clause of } F \\ l \textit{ is undefined in } M \end{cases}$$

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 :

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

Backtrack :

$$M\, l^{\mathsf{d}}\, N \parallel F, C \implies M\, \neg l \parallel F, C \quad \textbf{if} \begin{cases} M\, l^{\mathsf{d}}\, N \models \neg C \\ N \textit{ contains no decision literals} \end{cases}$$

One can use the transition system Cl for deciding the satisfiability of an input CNF F by simply generating an arbitrary derivation $\emptyset \parallel F \Longrightarrow_{Cl} \ldots \Longrightarrow_{Cl} S_n$, where S_n is a final state with respect to Cl. Such derivations are always finite, and (i) F is unsatisfiable if, and only if, the final state S_n is *FailState*, and (ii) if S_n is of the form $M \parallel F$ then M is a model of F.

These rules speak for themselves, providing a classical depth-first search with backtracking, where the Decide rule represents a case split: an undefined literal l is chosen from F, and added to M. The literal is annotated as a *decision literal*, to denote that, if $M\ l$ cannot be extended to a model of F, then (by Backtrack) still the other possibility $M\ \neg l$ must be explored. In the following, if M is a sequence of the form $M_0\ l_1\ M_1\ \ldots\ l_k\ M_k$, where the l_i are all the decision literals in M, then the literals of each $l_i\ M_i$ are said to *belong to decision level i*.

Example 2. In the following derivation, to improve readability we have denoted atoms by natural numbers, negation by overlining, and written decision literals in **bold** font:

$$
\begin{array}{rcllllllll}
\emptyset & \parallel & \overline{1}\vee\overline{2}, & 2\vee 3, & \overline{1}\vee\overline{3}\vee 4, & 2\vee\overline{3}\vee\overline{4}, & 1\vee 4 & \Longrightarrow_{Cl} & (\mathsf{Decide}) \\
\mathbf{1} & \parallel & \overline{1}\vee\overline{2}, & 2\vee 3, & \overline{1}\vee\overline{3}\vee 4, & 2\vee\overline{3}\vee\overline{4}, & 1\vee 4 & \Longrightarrow_{Cl} & (\mathsf{UnitPropagate}) \\
\mathbf{1}\ \overline{2} & \parallel & \overline{1}\vee\overline{2}, & 2\vee 3, & \overline{1}\vee\overline{3}\vee 4, & 2\vee\overline{3}\vee\overline{4}, & 1\vee 4 & \Longrightarrow_{Cl} & (\mathsf{UnitPropagate}) \\
\mathbf{1}\ \overline{2}\ 3 & \parallel & \overline{1}\vee\overline{2}, & 2\vee 3, & \overline{1}\vee\overline{3}\vee 4, & 2\vee\overline{3}\vee\overline{4}, & 1\vee 4 & \Longrightarrow_{Cl} & (\mathsf{UnitPropagate}) \\
\mathbf{1}\ \overline{2}\ 3\ 4 & \parallel & \overline{1}\vee\overline{2}, & 2\vee 3, & \overline{1}\vee\overline{3}\vee 4, & 2\vee\overline{3}\vee\overline{4}, & 1\vee 4 & \Longrightarrow_{Cl} & (\mathsf{Backtrack}) \\
\overline{1} & \parallel & \overline{1}\vee\overline{2}, & 2\vee 3, & \overline{1}\vee\overline{3}\vee 4, & 2\vee\overline{3}\vee\overline{4}, & 1\vee 4 & \Longrightarrow_{Cl} & (\mathsf{UnitPropagate}) \\
\overline{1}\ 4 & \parallel & \overline{1}\vee\overline{2}, & 2\vee 3, & \overline{1}\vee\overline{3}\vee 4, & 2\vee\overline{3}\vee\overline{4}, & 1\vee 4 & \Longrightarrow_{Cl} & (\mathsf{Decide}) \\
\overline{1}\ 4\ \overline{\mathbf{3}} & \parallel & \overline{1}\vee\overline{2}, & 2\vee 3, & \overline{1}\vee\overline{3}\vee 4, & 2\vee\overline{3}\vee\overline{4}, & 1\vee 4 & \Longrightarrow_{Cl} & (\mathsf{UnitPropagate}) \\
\overline{1}\ 4\ \overline{\mathbf{3}}\ 2 & \parallel & \overline{1}\vee\overline{2}, & 2\vee 3, & \overline{1}\vee\overline{3}\vee 4, & 2\vee\overline{3}\vee\overline{4}, & 1\vee 4 & & \text{Final state:} \\
 & & & & & & & & \quad\text{model found.}
\end{array}
$$
□

The Davis-Putnam procedure [DP60] was originally presented as a two-phase proof-procedure for first-order logic. The unsatisfiability of a formula was to be proved by first generating a suitable set of ground instances which then, in the second phase, were shown to be propositionally unsatisfiable.

Subsequent improvements, such as the Davis-Logemann-Loveland procedure of [DLL62], mostly focused on the propositional phase. What most authors nowadays call the *DPLL Procedure* is a satisfiability procedure for propositional logic based on this propositional phase. Originally, this procedure amounted to the depth-first search algorithm with backtracking modeled by our Classical DPLL system.

2.2 Modern DPLL Procedures

The major modern DPLL-based SAT solvers do not implement the Classical DPLL system. For example, due to efficiency reasons the pure literal rule is normally only used as a preprocessing step, and hence we will not consider this rule in the following. Moreover, instead of Backtrack a more general Backjump rule is considered, of which Backtrack is a particular case.

Definition 3. *The* Basic DPLL system *is the four-rule transition system B consisting of UnitPropagate, Decide, Fail, and the following Backjump rule:*

Backjump :

$$M\ l^{\mathsf{d}}\ N \parallel F, C \implies M\ l' \parallel F, C \quad \textbf{if} \begin{cases} M\ l^{\mathsf{d}}\ N \models \neg C, \text{ and there is} \\ \text{some clause } C' \vee l' \text{ such that:} \\ \quad F, C \models C' \vee l' \text{ and } M \models \neg C', \\ \quad l' \text{ is undefined in } M, \text{ and} \\ \quad l' \text{ or } \neg l' \text{ occurs in } F \text{ or in } M\ l^{\mathsf{d}}\ N \end{cases}$$

We call the clause $C' \vee l'$ in Backjump a backjump *clause.*

Example 4. The aim of this Backjump rule is to generalize backtracking by a better analysis of why the so-called *conflicting* clause C is false. Standard backtracking reverses the *last* decision, and adds it as a unit to the previous decision level. Backjumping frequently allows one to add a new unit literal to a decision level that is lower than the previous level. Consider:

$$\begin{array}{rcll}
\emptyset & \parallel & \overline{1}\vee 2,\ \ \overline{3}\vee 4,\ \ \overline{5}\vee\overline{6},\ \ 6\vee\overline{5}\vee\overline{2} & \Longrightarrow_{\mathrm{B}} \ \ (\mathsf{Decide}) \\
\mathbf{1} & \parallel & \overline{1}\vee 2,\ \ \overline{3}\vee 4,\ \ \overline{5}\vee\overline{6},\ \ 6\vee\overline{5}\vee\overline{2} & \Longrightarrow_{\mathrm{B}} \ \ (\mathsf{UnitPropagate}) \\
\mathbf{1}\ 2 & \parallel & \overline{1}\vee 2,\ \ \overline{3}\vee 4,\ \ \overline{5}\vee\overline{6},\ \ 6\vee\overline{5}\vee\overline{2} & \Longrightarrow_{\mathrm{B}} \ \ (\mathsf{Decide}) \\
\mathbf{1}\ 2\ \mathbf{3} & \parallel & \overline{1}\vee 2,\ \ \overline{3}\vee 4,\ \ \overline{5}\vee\overline{6},\ \ 6\vee\overline{5}\vee\overline{2} & \Longrightarrow_{\mathrm{B}} \ \ (\mathsf{UnitPropagate}) \\
\mathbf{1}\ 2\ \mathbf{3}\ 4 & \parallel & \overline{1}\vee 2,\ \ \overline{3}\vee 4,\ \ \overline{5}\vee\overline{6},\ \ 6\vee\overline{5}\vee\overline{2} & \Longrightarrow_{\mathrm{B}} \ \ (\mathsf{Decide}) \\
\mathbf{1}\ 2\ \mathbf{3}\ 4\ \mathbf{5} & \parallel & \overline{1}\vee 2,\ \ \overline{3}\vee 4,\ \ \overline{5}\vee\overline{6},\ \ 6\vee\overline{5}\vee\overline{2} & \Longrightarrow_{\mathrm{B}} \ \ (\mathsf{UnitPropagate}) \\
\mathbf{1}\ 2\ \mathbf{3}\ 4\ \mathbf{5}\ \overline{6} & \parallel & \overline{1}\vee 2,\ \ \overline{3}\vee 4,\ \ \overline{5}\vee\overline{6},\ \ 6\vee\overline{5}\vee\overline{2} & \Longrightarrow_{\mathrm{B}} \ \ (\mathsf{Backjump}) \\
\mathbf{1}\ 2\ \overline{5} & \parallel & \overline{1}\vee 2,\ \ \overline{3}\vee 4,\ \ \overline{5}\vee\overline{6},\ \ 6\vee\overline{5}\vee\overline{2} &
\end{array}$$

Before the Backjump step, the clause $6\vee\overline{5}\vee\overline{2}$ is conflicting: it is false in $\mathbf{1}\ 2\ \mathbf{3}\ 4\ \mathbf{5}\ \overline{6}$. The reason for its falsity is the unit propagation 2 of the decision **1**, together with the decision **5** and its unit propagation $\overline{6}$. Therefore, one can infer that the decision **1** (and its unit propagation 2) is incompatible with the decision **5**. This is why the Backjump rule moves to the state $\mathbf{1}\ 2\ \overline{5}$.

Note that an application of Backtrack instead of Backjump would have given a state with first component $\mathbf{1}\ 2\ \mathbf{3}\ 4\ \overline{5}$, even though the decision level $\mathbf{3}\ 4$ is unrelated with the reasons for the falsity of $6\vee\overline{5}\vee\overline{2}$. Moreover, intuitively, the search state $\mathbf{1}\ 2\ \overline{5}$ reached after Backjump is more *advanced* than $\mathbf{1}\ 2\ \mathbf{3}\ 4\ \overline{5}$. This notion of "being more advanced" is formalized in Theorem 12 below. □

The Backjump rule makes progress in the search by reverting to a strictly lower decision level, but with the additional information given by the literal l' that is added to that level. Indeed, as it is proved below, the four rules of the Basic DPLL system (UnitPropagate, Decide, Fail, and Backjump) suffice for completeness. But in most modern DPLL implementations, in addition the backjump clause $C' \vee l'$ is *added to the clause set* as a *learned* clause (*conflict-driven clause learning*). In Example 4, learning the clause $\overline{1}\vee\overline{5}$ will allow the application of UnitPropagate to any state whose assignment contains either 1 or 5. Hence, it will prevent any conflict caused by having both 1 and 5 in M. Indeed, reaching such *similar* conflicts frequently happens in industrial problems having some regular structure, and learning such lemmas has been shown to be very effective. Since a lemma is aimed at preventing future similar conflicts, when such conflicts are not very likely to be found again the lemma can be removed. In practice this is usually done if the *activity* of a lemma (e.g., the number of times it becomes a unit or a conflicting clause) has become low [ES03]. In order to model lemma learning and removal we consider the following system.

Definition 5. *The rules of* Learn *and* Forget *are the following ones:*

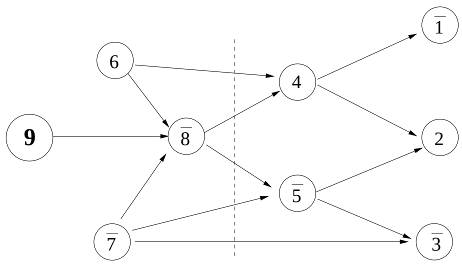
$$\begin{array}{llll} \textsf{Learn}: & & & \\ \quad M \parallel F & \Longrightarrow M \parallel F,\, C & \textbf{if} & \begin{cases} \textit{all atoms of } C \textit{ occur in } F \\ F \models C \end{cases} \\ \textsf{Forget}: & & & \\ \quad M \parallel F,\, C & \Longrightarrow M \parallel F & \textbf{if} & \left\{ F \models C \right. \end{array}$$

In any application step of these two rules, the clause C is said to be learned *and* forgotten*, respectively.*

Example 6. Assume a strategy that is followed in most state-of-the-art SAT solvers: (i) Decide is applied only if no other Basic DPLL rule is applicable, and (ii) after each application of Backjump, the backjump clause is learned. Consider a state of the form $M \parallel F$, where, among other clauses, F contains:

$$\overline{9}\lor\overline{6}\lor 7\lor\overline{8} \quad 8\lor 7\lor\overline{5} \quad \overline{6}\lor 8\lor 4 \quad \overline{4}\lor\overline{1} \quad \overline{4}\lor 5\lor 2 \quad 5\lor 7\lor\overline{3} \quad 1\lor\overline{2}\lor 3$$

and M is of the form: $\ldots 6 \ldots \overline{7} \ldots \mathbf{9}\ \overline{8}\ \overline{5}\ 4\ \overline{1}\ 2\ \overline{3}$. It is easy to observe how by six applications of UnitPropagate this state has been reached after the last decision $\mathbf{9}$. For example, $\overline{8}$ is implied by 9, 6, and $\overline{7}$, due to the leftmost clause $\overline{9}\lor\overline{6}\lor 7\lor\overline{8}$. The DPLL implementation stores the ordered sequence of propagated literals, each one of them together with the clause that caused it. In this state $M \parallel F$, the clause $1\lor\overline{2}\lor 3$ is conflicting, since M contains $\overline{1}$, 2 and $\overline{3}$. Now one can trace back the reasons for this conflicting clause. For example, the DPLL implementation knows that $\overline{3}$ was implied by $\overline{5}$ and $\overline{7}$, due to the clause $5\lor 7\lor\overline{3}$. The literal $\overline{5}$ was in turn implied by $\overline{8}$ and $\overline{7}$, and so on. In this way, working backwards from the conflicting clause, and in the reverse order in which each literal was propagated, one can build a *conflict graph*:



In the graph that is shown, the building process was stopped when the current decision literal **9** was reached, and hence **9** and the nodes belonging to earlier decision levels (in this example, literals 6 and $\overline{7}$) have no incoming arrows. For finding a backjump clause, it suffices to cut the graph into two parts. The first part must contain at least the literals with no incoming arrows. The second part must contain at least the literals with no outgoing arrows, i.e., the negated literals of the conflicting clause (in our example, $\overline{1}$, 2 and $\overline{3}$). It is not hard to see that in such a cut no model of F can satisfy all the literals whose outgoing edges are cut.

For instance, consider the cut indicated by the dotted line, where the literals with cut outgoing edges are $\overline{8}$, $\overline{7}$, and 6. Indeed, from these three literals by unit propagation using five clauses of F one can infer the negated literals of the conflicting clause. Hence, one can infer from F that $\overline{8}$, $\overline{7}$, and 6 cannot be simultaneously true, i.e., one can infer the clause $8\vee 7\vee\overline{6}$. In this case, it is an adequate backjump clause, that is, the clause $C' \vee l'$ in the definition of the Backjump rule, where the literal 8 plays the role of l'. Indeed, it allows one to backjump to the decision level of $\overline{7}$, adding 8 to it. After that, under our strategy the clause $8\vee 7\vee\overline{6}$ has to be learned, in order to explain, in future conflicts, the presence of 8 as a propagation from 6 and $\overline{7}$.

Such a cut produces an adequate backjump clause provided that only one of the literals with cut outgoing edges belongs to the current decision level. Then, this literal is called a *Unique Implication Point (UIP)* and it can play the role of l' in the backjump clause. It is not hard to argue that there is always at least one UIP, namely the current decision literal (which is **9** in our example). In practice one does not actually build the graph; it suffices to work backwards from the conflicting clause, maintaining only a *frontier* list of literals yet to be expanded, until reaching the *first UIP* (in our example, $\overline{8}$) [MSS99,ZMMM01,GN02]. This can also be seen as a resolution process, until reaching a clause with only one literal of the current decision level (in our example, the literal 8 in the clause $8\vee 7\vee\overline{6}$). In our example, the clause $8\vee 7\vee\overline{6}$ is obtained by resolution by resolving on the conflicting clause the literals $\overline{3}, 2, \overline{1}, \overline{4}$ and 5, i.e., in the reverse order they were propagated, with the clauses that caused their propagation:

$$\dfrac{8\vee 7\vee \overline{5} \qquad \dfrac{\overline{6}\vee 8\vee 4 \qquad \dfrac{\overline{4}\vee\overline{1} \qquad \dfrac{\overline{4}\vee 5\vee 2 \qquad \dfrac{5\vee 7\vee\overline{3} \qquad 1\vee\overline{2}\vee 3}{5\vee 7\vee 1\vee\overline{2}}}{\overline{4}\vee 5\vee 7\vee 1}}{\overline{4}\vee 5\vee 7}}{\overline{6}\vee 8\vee 5\vee 7}}{8\vee 7\vee\overline{6}}$$

Some provers such as Siege also learn some of the intermediate clauses in such resolution derivations [Rya04]. □

State-of-the art SAT-solvers [MMZ+01,GN02,ES03,Rya04] essentially apply Abstract DPLL with Learning using efficient implementation techniques for UnitPropagate (e.g., watching two literals for unit propagation [MMZ+01]), and heuristics for selecting the decision literal when applying the Decide rule. In addition, modern DPLL implementations *restart* the DPLL procedure whenever the search is not making enough progress according to some measure. The rationale behind this idea is that upon each restart, the newly learned lemmas will lead the heuristics for Decide to behave differently, and hopefully cause the procedure to explore the search space in a more compact way.

The combination of learning and restarts has been shown to be powerful not only in practice, but also from the theoretical point of view. Essentially, any Basic DPLL derivation to *FailState* is equivalent to *tree-like* refutation by resolution. But for some classes of problems tree-like proofs are always exponentially larger than the smallest *general*, i.e., DAG-like, resolution ones [BEGJ00]. The good news is that DPLL with learning and restarts becomes again equivalent to general resolution with respect to such notions of proof complexity [BKS03].

Definition 7. *The* DPLL system with learning and restarts*, denoted by L, consists of the four transition rules of the Basic DPLL system, the* Learn *and* Forget *rules and the following* Restart *rule:*

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

2.3 Correctness of Modern DPLL Systems

Deciding the satisfiability of an input formula F will be done by generating an arbitrary derivation of the form $\emptyset \parallel F \implies_{\mathrm{L}} \dots \implies_{\mathrm{L}} S_n$ such that S_n is *final with respect to the Basic DPLL system* (note that one cannot aim at reaching final states with respect to the DPLL system with learning, since, e.g., tautologies like $p \vee \neg p$ can be learned or forgotten in all states but *FailState*).

Building such derivations is practical because for all rules their applicability is easy to check, and such derivations are always finite if one never applies infinitely many consecutive Learn and Forget steps, and Restart is applied with increasing periodicity. Then, one always reaches a state S_n that is final with respect to the Basic DPLL system, and a final state is moreover easily recognizable as such, because it is either *FailState* or else it is of the form $M \parallel F$ where all literals of F are defined in M and there is no conflicting clause. Then, moreover, (i) F is unsatisfiable if, and only if, S_n is *FailState*, and (ii) if S_n is of the form $M \parallel F'$ then M is a model of F.

The following three lemmas are the key to proving these results (see [NOT05] for details). The first one states some easy invariants that are preserved under rule application. Proving the second one essentially involves the construction of an adequate backjump clause for showing that Backjump applies, which is less simple. From these two lemmas, the third one, stating properties of final states, is not hard to obtain.

Lemma 8. *Assume* $\emptyset \parallel F \implies_{\mathrm{L}}^{*} M \parallel G$ *. Then* G *is logically equivalent to* F*. If* M *is of the form* $M_0\ l_1\ M_1\ \dots\ l_n\ M_n$*, where* $l_1, \dots, l_n$ *are all the decision literals of* M*, then* $F, l_1, \dots, l_i \models M_i$ *for all* i *in* $0 \dots n$*.*

Lemma 9. *Assume that $\emptyset \parallel F \Longrightarrow_{\mathrm{L}}^{*} M \parallel F'$ and that $M \models \neg C$ for some clause C in F'. Then either* Fail *or* Backjump *applies to $M \parallel F'$.*

Lemma 10. *If $\emptyset \parallel F \Longrightarrow_{\mathrm{L}}^{*} S$, and S is final with respect to Basic DPLL, then S is either FailState, or it is of the form $M \parallel F'$, where*
(i) all literals of F' are defined in M, and
(ii) there is no clause C in F' such that $M \models \neg C$, and
(iii) M is a model of F.

Theorem 11. *If $\emptyset \parallel F \Longrightarrow_{\mathrm{L}}^{*} S$ where S is final w.r.t. Basic DPLL, then*

1. *S is FailState if, and only if, F is unsatisfiable.*
2. *If S is of the form $M \parallel F'$ then M is a model of F.*

Proof. For Property 1, if S is *FailState* it is because there is some state $M \parallel F'$ such that $\emptyset \parallel F \Longrightarrow_{\mathrm{L}}^{*} M \parallel F' \Longrightarrow_{\mathrm{L}} \mathit{FailState}$. By the definition of the Fail rule, there is no decision literal in M and there is a clause C in F' such that $M \models \neg C$. Since F and F' are equivalent by Lemma 8, we have that $F \models C$. However, if $M \models \neg C$, by Lemma 8 then also $F \models \neg C$, which implies that F is unsatisfiable. For the right-to-left implication, if S is not *FailState* it has to be of the form $M \parallel F'$. But then, by Lemma 10, M is a model of F and hence F is satisfiable. For Property 2, if S is $M \parallel F'$ then, again by Lemma 10, M is a model of F. □

The soundness and completeness results of Theorem 11 can be applied if one can ensure that a final state with respect to Basic DPLL is eventually reached. This is usually done in practice by periodically increasing the minimal number of Basic DPLL steps between each pair of restart steps. Also, one should not apply infinitely many consecutive Learn and Forget steps (for example, learning and forgetting the same clause all the time), a condition that is weak and easily enforced. In fact, Learn is typically only applied together with Backjump in order to learn the corresponding backjump clause. This is formalized below.

Theorem 12. *Any derivation $\emptyset \parallel F \Longrightarrow S_1 \Longrightarrow \ldots$ by the transition system L extended with the* Restart *rule is finite if (i) it contains only finitely many consecutive* Learn *and* Forget *steps, and (ii) between every two* Restart *steps there are more steps by Basic DPLL than between the previous two* Restart *steps.*

Proof. (See [NOT05] for details.) The four basic rules can be shown terminating by a well-founded ordering $\succ$ that considers only the first component M of states of the form $M \parallel F$. The ordering is lexicographic. It considers M more advanced than M' (i.e., $M' \succ M$) if M has more literals at decision level 0 than M', or both have the same number of literals at level 0 and M has more literals at level 1, etc. If D is an infinite derivation fulfilling the requirements, then in a subderivation of D without Restart steps, at each step either this first component decreases with respect to $\succ$ (by the Basic DPLL steps) or it remains equal (by the Learn and Forget steps). Therefore, since there are no infinitely many consecutive Learn and Forget steps, there must be infinitely many Restart steps in D. Also, if between two states there is at least one Basic DPLL step, these states do not have the same first component. Therefore, if N denotes the (fixed, finite) number of different first components of states that exist for the given finite set of symbols, there are no subderivations with more than N Basic DPLL steps between two Restart steps. This contradicts the fact that there are inifinitely many Restart steps if Restart has increasing periodicity. in D. $\square$

3 Abstract DPLL Modulo Theories

Here we consider the same definitions and notations given in Section 2 except that here the set P over which formulas are built is a fixed finite set of *ground* (i.e., variable-free) first-order atoms (instead of propositional symbols). In addition to these propositional notions, a *theory* T is a set of closed first-order formulas that is satisfiable in the first-order sense. A formula F is *T-satisfiable* or *T-consistent* if $F \wedge T$ is satisfiable in the first-order sense. Otherwise, it is called *T-unsatisfiable* or *T-inconsistent*. As before, a partial assignment M will also be seen as a conjunction and hence as a formula. If M is a T-consistent partial assignment and F is a formula such that $M \models F$, i.e., M is a (propositional) model of F, then we say that *M is a T-model of F*. The SMT problem for a theory T is the problem of determining, given a formula F, whether F is T-satisfiable, or, equivalently, whether F has a T-model. As usual in SMT, here we only consider the T-satisfiability of *ground* (and hence quantifier-free) CNF formulas F. But note that F may contain constants that are free in T, which, as far as satisfiability is concerned, can equivalently be seen as existential variables. We will consider here only theories T such that the T-satisfiability of conjunctions of such ground literals is decidable, and a decision procedure for doing so is called a *T-solver*. If F and G are formulas, then *F entails G in T*, written $F \models_T G$, if $F \wedge \neg G$ is T-inconsistent. If $F \models_T G$ and $G \models_T F$, we say that F and G are *T-equivalent*.

3.1 An informal presentation of SMT procedures

In the so-called *eager* approach to SMT, the input formula is translated in a single satisfiability-preserving step into a propositional CNF formula which is then checked by a SAT solver for satisfiability (see, e.g., [BGV01,BV02,Str02]). Sophisticated ad-hoc translations have been developed for several theories, but still, on many practical problems the translation process or the SAT solver run out of time or memory (see [dMR04]), and the alternative techniques explained below are usually orders of magnitude faster.

As an alternative to the eager approach, one can use a T-solver for deciding the satisfiability of conjunctions of theory literals. Then, a decision procedure for SMT is easily obtained by converting the formula into disjunctive normal form (DNF) and using the T-solver for checking whether there is at least one conjunction which is satisfiable. However, the exponential blowup due to the conversion into DNF makes this approach too inefficient. Therefore, a large amount of recent research involves the combination of the strengths of specialized T-solvers with the strengths of state-of-the-art SAT solvers for dealing with the boolean structure of the formulas. One such an approach, which has been widely used in the last few years is usually referred to as the *lazy* approach [ACG00,FORS01,ABC$^+$02,BDS02,dMR02,FJOS03,ACGM04], [BCLZ04]. It initially considers each atom occurring in a formula F to be checked for satisfiability simply as a propositional symbol, i.e., it "forgets" about the theory T. Then it sends the formula to a SAT solver. If the SAT solver reports propositional unsatisfiability, then F is also T-unsatisfiable. If the SAT solver returns a propositional model of F, then this model (a conjunction of literals) is checked by a T-solver. If it is found T-satisfiable then it is a T-model of F. Otherwise, the T-solver builds a ground clause, called a *theory lemma*, a clause C such that $\emptyset \models_T C$, precluding that model. This lemma is added to F and the SAT solver is started again. This process is repeated until the SAT solver finds a T-satisfiable model or returns unsatisfiable.

Example 13. Assume we are deciding the satisfiability of a large EUF formula, i.e., the background theory T is equality, and assume that the model M found by the SAT solver contains, among many others, the literals: $b = c$, $f(b) = c$, $a \neq g(b)$, and $g(f(c)) = a$. Then the T-solver detects that M is not a T-model, since $\quad b = c \quad \wedge \quad f(b) = c \quad \wedge \quad g(f(c)) = a \quad \models_T \quad a = g(b)$. Therefore, the lazy procedure has to be restarted after the corresponding theory lemma has been added to the clause set. In principle, one can take as theory lemma simply the negation of M, that is, the disjunction of the negations of all the literals in M. However, this clause may therefore have thousands of literals, and the lazy approach will behave much more efficiently if the T-solver is able to generate a small *explanation* of the T-inconsistency of M, which in this example could be the clause $\quad b \neq c \quad \vee \quad f(b) \neq c \quad \vee \quad g(f(c)) \neq a \quad \vee \quad a = g(b)$. □

The lazy approach is quite flexible: it can easily combine any SAT solver with any T-solver. Moreover, if the SAT solver used by the lazy SMT procedure is based on DPLL, several refinements exist that make it much more efficient:

Incremental T-solver. The T-consistency of the model can be checked incrementally, while the model is being built by the DPLL procedure, i.e., without delaying the check until a propositional model has been found. This can save a large amount of useless work. Currently, most SMT implementations work with incremental T-solvers. The idea was already mentioned in [ABC$^+$02] under the name of *early pruning* and in [Bar03] under the name of *eager notification*.

On-line SAT solver. When a T-inconsistency is detected by the incremental T-solver, one can ask the DPLL procedure simply to backtrack to the last point where the assignment was still T-consistent, instead of restarting the search from scratch. If the current DPLL state is of the form $M\, l\, M' \parallel F$, and M is the maximal T-consistent prefix of $M\, l\, M'$, then the DPLL procedure can, for instance, backjump to $M \neg l \parallel F$. On-line SAT solvers (in combination with incremental T-solvers) are nowadays common in SMT implementations.

Theory propagation. In the approach presented so far, the T-solver provides information only *after* a T-inconsistent partial assignment has been generated.

In this sense, the T-solver is used only to *validate* the search a posteriori, not to *guide* it a priori. In order to overcome this limitation, the T-solver can also be used in a given DPLL state $M \parallel F$ to detect literals l ocurring in F such that $M \models_T l$, allowing the DPLL procedure to move to the state $M\,l \parallel F$. This is called *theory propagation*. It was first mentioned in [ACG00] under the name of *forward checking simplification*; however, it was believed to be very expensive. Since T-solvers were not designed to support it, it was simply implemented by sending $\neg l$ to the T-solver, and, if this made the model T-inconsistent, then inferring l. The real effectiveness of theory propagation has become demonstrated in our DPLL(T) approach [GHN⁺04,NO05b], using efficient T-solvers for it.

Exhaustive Theory Propagation. For some theories it even pays off, for every state $M \parallel F$, to eagerly detect and propagate *all* literals l ocurring in F such that $M \models_T l$ [NO05b]. Then, in every state $M \parallel F$ the model M will be T-consistent, and hence the T-solver will never detect any T-inconsistencies. Similarly, theory lemma learning becomes useless if exhaustive theory propagation is applied, because any unit propagation from a theory lemma will already be immediately obtained as a theory propagation. For some logics, such as, e.g., Difference Logic, exhaustive theory propagation can give several orders of magnitude of speedup (see Section 5).

3.2 Abstract DPLL Modulo Theories

In this section we formalize the different enhancements of the lazy approach to Satisfiability Modulo Theories. This will be done by adapting the abstract DPLL framework for the propositional case presented in the previous section. Here Learn, Forget and $\mathsf{Backjump}$ are slightly modified in order to work modulo theories: in these rules, entailment between formulas now becomes entailment in T:

Definition 14. *The rules T-Learn, T-Forget and T-$\mathsf{Backjump}$ are:*

T-Learn :

$$M \parallel F \quad \Longrightarrow \quad M \parallel F, C \quad \textbf{if} \begin{cases} \textit{every atom of } C \textit{ occurs in } F \textit{ or in } M \\ F \models_T C \end{cases}$$

T-Forget :

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

T-$\mathsf{Backjump}$:

$$M\ l^{\mathsf{d}}\ N \parallel F, C \Longrightarrow M\ l' \parallel F, C \ \textbf{if} \begin{cases} M\ l^{\mathsf{d}}\ N \models \neg C, \textit{ and there is} \\ \textit{some clause } C' \vee l' \textit{ such that:} \\ \quad F, C \models_T C' \vee l' \textit{ and } M \models \neg C', \\ \quad l' \textit{ is undefined in } M, \textit{ and} \\ \quad l' \textit{ or } \neg l' \textit{ occurs in } F \textit{ or in } M\ l^{\mathsf{d}}\ N \end{cases}$$

Modeling the naive lazy approach. Each time a state $M \parallel F$ is reached that is final with respect to Decide, Fail, UnitPropagate, and T-Backjump, i.e., final in a similar sense as in the previous section, M can be T-consistent or not. If it is, then M is indeed a T-model of F. If it is not, then there exists a subset $\{l_1, \dots, l_n\}$ of M such that $\emptyset \models_T \neg l_1 \vee \dots \vee \neg l_n$. By one T-Learn step, this theory lemma $\neg l_1 \vee \dots \vee \neg l_n$ can be learned and then Restart can be applied. If these theory lemmas are never removed by the T-Forget rule, this stategy is terminating under the same restrictions as stated in the previous section on T-Learn, T-Forget, and Restart, and it is also sound and complete: the initial formula is T-unsatisfiable iff, the final state is *FailState*, and otherwise a T-model has been found.
Modeling the lazy approach with an incremental T-solver. Assume the incremental T-solver detects that a (not necessarily final) state $M \parallel F$ has been reached such that M is T-inconsistent. Then, as in the naive lazy approach, there exists a subset $\{l_1, \dots, l_n\}$ of M such that $\emptyset \models_T \neg l_1 \vee \dots \vee \neg l_n$. This theory lemma is then learned, reaching the state $M \parallel F,\ \neg l_1 \vee \dots \vee \neg l_n$. As in the previous case, then Restart can be applied and the same results apply.
Modeling the lazy approach with an incremental T-solver and an on-line SAT solver. As in the previous case, if a T-inconsistency is detected a state $M \parallel F,\ \neg l_1 \vee \dots \vee \neg l_n$ is reached. But now instead of completely restarting, the procedure *repairs* the T-inconsistency of the partial model by exploiting the fact that $\neg l_1 \vee \dots \vee \neg l_n$ is a conflicting clause. Then, as before, if there is no decision literal in M then Fail applies, and otherwise T-Backjump applies. Even if always immediately after backjumping the theory lemma is forgotten, the termination, soundness and completeness results hold.
Modeling the previous refinements and theory propagation. This requires the following additional rule:

Definition 15. *The Theory Propagate rule is:*

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

The purpose of this rule is to prune the search by assigning a truth value to literals that are T-entailed by M. Below we prove that the results of termination, soundness, and completeness mentioned for the previous three lazy approaches also hold in combination with arbitrary applications of this rule.

Modeling the previous refinements and exhaustive theory propagation. Exhaustive theory propagation is modeled simply by assuming that Theory Propagate is applied eagerly. As a particular case of the previous refinement (arbirary applications of Theory Propagate), the aforementioned results remain true.

3.3 Correctness of Abstract DPLL Modulo Theories

Definition 16. *The Basic DPLL Modulo Theories system consists of the rules Decide, Fail, UnitPropagate, and T-Backjump.*

The Full DPLL system Modulo Theories*, denoted by* FT*, consists of the Basic DPLL Modulo Theories rules and the rules of* Theory Propagate*,* T-Learn*,* T-Forget*, and* Restart*.*

The proofs of the following results are structured in the same way as the ones given in Section 2.3 for the propositional case (see [NOT05] for details). As before, a decision procedure is any derivation by the given rules using a terminating strategy, and again we consider as final states, apart from *FailState*, the ones of the form $M \parallel F$ that are final with respect to the four rules of Basic DPLL Modulo Theories, but now in addition we require that the model M is T-consistent. We provide here only one additional property, showing that such final states can be effectively computed:

Property 17. If $\emptyset \parallel F \Longrightarrow^*_{\mathrm{FT}} M \parallel F'$ and M is T-inconsistent, then either there is a conflicting clause in $M \parallel F'$, or else T-Learn applies to $M \parallel F'$, generating a conflicting clause.

Theorem 18.

1. *If* $\emptyset \parallel F \Longrightarrow^*_{\mathrm{FT}}$ *FailState then* F *is* T*-unsatisfiable.*
2. *If* $\emptyset \parallel F \Longrightarrow^*_{\mathrm{FT}} S$ *where* S *is final with respect to Basic DPLL modulo theories and* M *is* T*-consistent, then* M *is a* T*-model of* F*.*

Theorem 19 (Termination). *Any derivation* $\emptyset \parallel F \Longrightarrow_{\mathrm{FT}} S_1 \Longrightarrow_{\mathrm{FT}} \ldots$ *by the Full DPLL system modulo theories is finite, if it contains only finitely many consecutive* T-Learn *and* T-Forget *steps, and between every two* Restart *steps, either there are more steps by Basic DPLL Modulo Theories than between the previous two* Restart *steps, or else a new clause has been learned that is never forgotten in* D*.*

4 The DPLL(T) approach

In this section we shortly describe the DPLL(T) approach for SAT Modulo Theories [GHN+04,NO05b]. It is based on a general DPLL engine, called DPLL(X), that is not dependent on any particular theory T. Instead, it is parameterized by a solver for a theory T of interest. A system DPLL(T) for deciding the satisfiability of CNF formulas in a theory T is produced by instantiating the parameter X with a module $Solver_T$ that can handle conjunctions of literals in T. The basic idea is similar to the $CLP(X)$ scheme for constraint logic programming: provide a clean and modular, but at the same time efficient, integration of specialized theory solvers within a general-purpose engine, in our case one based on DPLL.

The concrete DPLL(T) scheme and its architecture and implementation presented here combine the advantages of the eager and lazy approaches to SMT. On the one hand, experiments for several different theories reveal that, as soon as the theory predicates start playing a significant role in the formula, our DPLL(T) approach outperforms all others. On the other hand, DPLL(T) has the flexibility of the lazy approaches: more general logics can be dealt with by simply plugging in other solvers into our general DPLL(X) engine, provided that these solvers conform to a minimal interface.

Here we describe two versions of the DPLL(T) approach, namely with and without exhaustive theory propagation. For the first case, in [NO05b] an efficient exhaustive solver for *difference logic* is described. For some other logics, such as the logic of Equality with Uninterpreted Functions (EUF, see Example 13), exhaustive theory propagation is not the best DPLL(T) approach. Our experiments with EUF revealed that detecting exhaustively all *negative* equality consequences is very expensive, whereas all positive equalities can be propagated efficiently by means of a congruence closure algorithm [DST80]. In [NO03] a modern incremental, backtrackable congruence closure algorithm for this purpose is described, and progressively more efficient ways of retrieving explanations in this context are described in [dMRS04,ST05,NO05c].

4.1 DPLL(T) with exhaustive theory propagation

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

- At each state $M \parallel F$, both DPLL(X) and Solver_T are aware of the current M. Each time DPLL(X) communicates to Solver_T that a literal l is added to M, (e.g., due to UnitPropagate or to Decide), Solver_T answers with the list of *all* literals of the input formula that are new T-consequences. Then, for each one of these consequences, Theory Propagate is immediately applied by DPLL(X). Note that hence M never becomes T-inconsistent.
- If Theory 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 T-Backjump if a conflicting clause is detected. T-Backjump works as explained in Example 6, but there is a difference: a literal l at a node in the graph can now also be due to an application of Theory Propagate. Hence, building the graph requires that Solver_T must be able to recover a (preferrably small) subset of literals of M that T-entailed l. This is done by the $\mathit{Explain}(l)$ operation provided by Solver_T. It is the same operation as for providing explanations in the lazy approach, cf. Example 13.
- Immediately after each T-Backjump application, the T-Learn rule is applied for learning the backjump clause. This clause is always a T-consequence of the current formula. As explained in Subsection 3.1 for exhaustive theory propagation, theory lemmas (clauses C such that $\emptyset \models_T C$) are not learned, since this is useless.
- After each backjump has taken place in DPLL(X), it tells Solver_T how many literals of the partial interpretation have been unassigned, which allows Solver_T to undo them.
- In our current implementation, DPLL(X) applies Restart when certain system parameters reach some prescribed limits, such as the number of conflicts or lemmas, the number of new units derived, etc.

- In our current implementation, T-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 Theory Propagate, UnitPropagate, Fail or T-Backjump is applicable. We currently use a heuristic for chosing the decision literal as in BerkMin [GN02].

4.2 DPLL(T) with non-exhaustive theory propagation

- Each time DPLL(X) adds a literal l to M, $Solver_T$ either indicates that M has become T-inconsistent, or, otherwise, it returns a (possibly incomplete) list of T-consequences to which Theory Propagate is immediately applied by DPLL(X) (as in the exhaustive case). T-inconsistencies are treated by DPLL(X) as described in Subsection 3.2 for modeling with an on-line SAT solver: if there is a subset $\{l_1, \ldots, l_n\}$ of M that becomes T-inconsistent by adding l to it, the corresponding theory lemma $\neg l_1 \vee \ldots \vee \neg l_n \vee \neg l$ is learned, and used as a backjump clause in a T-Backjump step.
- As before, if Theory Propagate is not applicable, then UnitPropagate is eagerly applied by DPLL(X), and Fail or T-Backjump are applied if a conflicting clause C is detected. After each T-Backjump application, $Solver_T$ is notified for unassigning literals, and T-Learn is applied for learning the backjump clause. Also Decide (only lazily) and Restart are applied as before.
- T-Forget is also applied as in the exhaustive case, but in this case among the (less active) lemmas that are removed there are also theory lemmas.

5 The BarcelogicTools

In this section we describe BarcelogicTools, a set of logic-based tools developed by our research group in Barcelona. The development of the BarcelogicTools is funded by the Spanish Ministry of Education and Science (TIN2004-03382), as well as by several private sources. The intended applications of BarcelogicTools range from hardware and software verification to industrial combinatorial optimization problems (planning, scheduling). Most of the tools are built around a state-of-the-art SAT solver, and there is also a DPLL(X) engine and a number of theory solvers that can be combined forming different DPLL(T) systems.

5.1 SMT inside BarcelogicTools

Currently, BarcelogicTools supports difference logic over the integers or the reals, equality with uninterpreted function symbols (EUF) and the interpreted functions symbols *predecessor* and *successor*, or combinations of these theories. More theory solvers for, e.g., linear integer and real arithmetic, the theory of arrays, and bit vectors are under development.

The system is written in C. Apart from the parser and the CNF translator, three are the main components of the system.

1. Its DPLL(X) engine has some 3500 lines of source code. It is based on the DPLL procedure and implements state-of-the-art techniques such as the two-watched literal scheme, 1UIP learning scheme and VSIDS-like decision heuristics, but does not present any significant novelty wrt. state-of-the-art SAT solvers.
2. The solver for EUF (some 4000 lines) is an extension of a congruence closure algorithm. Apart from determining the satisfiability of a given set of equalities and disequalities E, it can detect that *some* literals in the original formula are entailed by E. In addition, for each such literal the solver can compute a small subset of E of which the literal is already a logical consequence. More details can be found at [NO03,NO05c].
3. The solver for difference logic (1400 lines) can be seen an extension of a shortest-path algorithm aimed at determining, given a consistent set of difference constraints S, *all* literals in the original formula that are logically entailed by S. For each of these consequences, the solver can compute a minimal (wrt set inclusion) subset of S from which the literal is also entailed. For further details see [NO05b].

The effectivity of our approach was shown at the 2005 SMT Competition [BdMS05]. A large collection of benchmarks (around 1300) coming from diverse areas such software and hardware verification, bounded model checking, finite model finding, or scheduling were classified, according to the underlying theory or to some syntactic restrictions, into the 7 divisions of which the competition consisted. For each division, around 50 benchmarks were randomly chosen and given to each entrant with a time limit of 10 minutes per benchmark.

One single version of BarcelogicTools, our DPLL(T) implementation as described in Section 4, entered (and won) all four divisions for which it had a theory solver: EUF, IDL and RDL (integer and real difference logic), and UFIDL (combining EUF and IDL). The same Among the competitors were well-known SMT solvers like SVC [BDL96], CVC [BDS02], CVC-Lite [BB04], MathSAT [BBA$^+$05] and the very recent successors of ICS [FORS01], called Yices (by Leonardo de Moura) and Simplics (by Dutertre and de Moura). For each division, the results of the best three systems are given in the following table, where **Time** is the total time *for the solved problems*:

	top-3 systems	**# Problems solved**	**Time (secs.)**
EUF (50 problems):	BarcelogicTools	39	1758.2
	Yices	37	1801.4
	MathSAT	33	2186.2
RDL (50 pbms.):	BarcelogicTools	41	940.8
	Yices	37	1868.0
	MathSAT	37	2608.0
IDL (51 pbms.):	BarcelogicTools	47	1131.2
	Yices	47	1883.2
	MathSAT	46	1295.4
UFIDL (49 pbms.):	BarcelogicTools	45	305.2
	Yices	36	1989.8
	MathSAT	22	1055.5

5.2 Ad-Hoc Theory Combination and UFIDL

Perhaps the most remarkable results obtained by BarcelogicTools in the SMT Competition are the ones for the UFIDL division, where problems contain both uninterpreted functions and difference logic atoms, interpreted with respect to a background theory T of the integers. That is, atoms can be equalities $s = t$, or atoms of the form $s - t \leq k$, where s and t are ground terms built over uninterpreted symbols, and k is a concrete integer (apart from $\leq$, also $>$ may appear).

Many general results exist for the modular combination of decision procedures, à la Shostak, or à la Nelson-Oppen [Sho84,NO79]. But we believe that for certain classes of problems it is better to apply a more ad-hoc combination of theories. One particular example appears to be this combination of EUF and IDL.

Our procedure proceeds as follows. It first checks whether the input formula contains some ordering predicate ($\leq$ or $<$).

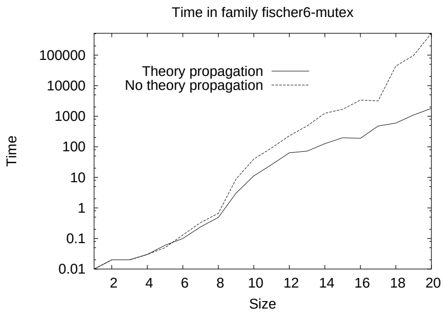
- If this is the case, first all function symbols are removed by means of Ackermann's reduction [Ack54]: for each pair of occurrences in the formula of terms of the form $f(s_1, \ldots, s_n)$ and $f(t_1, \ldots, t_n)$, a monotonicity clause
$$s_1 = t_1 \wedge \ldots \wedge s_n = t_n \longrightarrow f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)$$
is added. After that, the equality predicate can be encoded as an equivalence relation (i.e., not any more as a congruence relation). This can be done by simply considering $s = t$ as a difference logic atom (e.g., as $s \leq t \wedge t \leq s$), and hence only a theory solver for difference logic needs to be used.
- If the input formula contains no ordering predicates $\leq$ or $<$, an EUF solver using the congruence closure algorithm of [NO03] is used. Its extension with *integer offsets* for dealing with the symbols *predecessor* and *successor* (also described in [NO03]) allows for expressing literals of the form $s - t = k$ as equalities $s = t + k$.

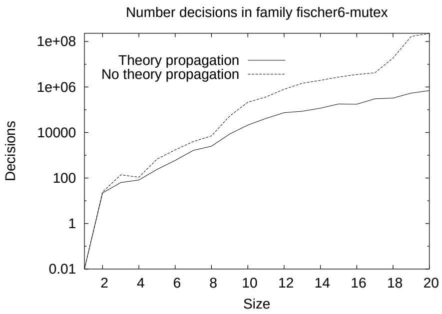
Even if there are no ordering predicates, if the number of function symbols is reasonably small it is sometimes still useful to add the monotonicity clauses of Ackermann's transformation. The reason is that it allows one to detect some propagations of negative equalities that would remain undetected in the non-exhaustive theory propagation approach used by BarcelogicTools for EUF. More precisely, it is not detected in general that $f(a) \neq f(b)$ implies $a \neq b$, which will be detected in the presence of the monotonicity clause $a \neq b \vee f(a) = f(b)$.

5.3 The role of Theory propagation in BarcelogicTools

In our experience, the overhead produced by theory propagation is usually compensated by a significant reduction of the search space. In [GHN+04] we already gave extensive experimental results showing its effectivity inside our DPLL(T) approach for EUF logic, and in [NO05b] a large amount of experiments are discussed for difference logic, with additional emphasis on the good scaling properties. Hence it is not surprising that new SMT solvers such as Yices and MathSAT also apply theory propagation. In fact, the most recent versions of MathSAT include exactly our congruence closure algorithms with theory propagation and Explain [NO03,NO05c] for its EUF solver.

In the following two figures, BarcelogicTools with and without theory propagation is compared in terms of runtime (in seconds) and number of decisions on a typical real-world difference logic suite (fisher6-mutex) consisting of 20 problems of increasing size.





The figures show the typical behaviour on the larger problems: both the runtime and the number of decisions are orders of magnitude smaller in the version with theory propagation. In both cases the $DPLL(\mathrm{X})$ engine used was exactly the same, although in the exhaustive theory case some parts of the code never applied (e.g., theory lemma learning).

Of course, theory propagation may not pay off in certain specific problems where the theory plays an insignificant role, i.e., where reasoning is done almost entirely at the boolean level. Such situations can be detected on the fly by computing the percentage of conflicts which are produced in part due to theory propagation. If this number is very low, theory propagation can be switched off automatically in order to speed up the computation.

5.4 Comparison of BarcelogicTools with the Eager Approach

For completeness, we finally compare DPLL(T) with UCLID, the best-known tool implementing the eager translation approach to SMT [LS04]. Three typical series of benchmarks of difference logic are considered, coming from different methods for pipelined processor verification given in [MS05a,MS05b]. Results of runtimes in seconds (with one hour timeout) are given using Siege [Rya04] as the final SAT solver for UCLID, since it gave the best results.

	UCLID	**DPLL(T)**	**UCLID**	**DPLL(T)**	**UCLID**	**DPLL(T)**
6 stage	258	1	3596	5	19	1
7 stage	835	3	>3600	8	58	1
8 stage	3160	15	>3600	18	226	1
9 stage	>3600	23	>3600	18	664	1
10 stage	>3600	54	>3600	29	>3600	2

6 Conclusions

We have shown that the Abstract DPLL formalism introduced here can be very useful for understanding and formally reasoning about a large variety of DPLL-based procedures for SAT and SMT.

In particular, we have used it here for describing two variants of a new, efficient, and modular approach for SMT, called DPLL(T). New theories can be dealt with by DPLL(T) by simply plugging in new theory solvers, which must only be able to deal with *conjunctions* of theory literals and conform to a minimal and simple set of additional requirements.

Current work inside the BarcelogicTools concerns the development of more theory solvers, for, e.g., linear integer and real arithmetic, the theory of arrays, and bit vectors, as well as the development of other logic-related tools.

Also, a new $DPLL(X_1, \ldots, X_n)$ engine is being developed for automatically dealing with the combination of theories, i.e., essentially standard theory solvers for theories $T_1, \ldots, T_n$ can be used for obtaining a system $DPLL(T_1, \ldots, T_n)$. We aim at an approach for doing this in a way similar to the one of [BBC+05], but where part of the equality reasoning takes place inside the $DPLL(X_1, \ldots, X_n)$ engine.

References

[ABC+02] G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowicz, and R. Sebastiani. A SAT based approach for solving formulas over boolean and linear mathematical propositions. In *CADE-18*, LNCS 2392, pages 195–210, 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.

[ACGM04] Alessandro Armando, Claudio Castellini, Enrico Giunchiglia, and Marco Maratea. A SAT-based Decision Procedure for the Boolean Combination of Difference Constraints. In *7th International Conference on Theory and Applications of Satisfiability Testing(SAT 2004)*. LNCS, 2004.

[Ack54] Wilhelm Ackermann. *Solvable Cases of the Decision Problem*. Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1954.

[Alu99] Rajeev Alur. Timed automata. In Nicolas Halbwachs and Doron Peled, editors, *Proceedings of the 11th International Conference on Computer Aided Verification, CAV'99 (Trento, Italy)*, volume 1633 of *Lecture Notes in Computer Science*, pages 8–22. Springer, 199.

[Bar03] Clark W. Barrett. *Checking Validity of Quantifier-Free Formulas in Combinations of First-Order Theories*. PhD thesis, Stanford University, 2003.

[BB04] Clark W. Barrett and Sergey Berezin. CVC lite: A new implementation of the cooperating validity checker category b. In R. Alur and D. Peled, editors, *Proceedings of the 16th International Conference on Computer*

Aided Verification, CAV'04 (Boston, Massachusetts), volume 3114 of *Lecture Notes in Computer Science*, 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 *Tools and Algorithms for the Construction and Analysis of Systems, 11th Int. Conf., (TACAS)*, volume 3440 of *Lecture Notes in Computer Science*, pages 317–333, 2005.

[BBC+05] Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Tommi A. Junttila, Silvio Ranise, Peter van Rossum, and Roberto Sebastiani. Efficient satisfiability modulo theories via delayed theory combination. In *Int. Conf. on Computer Aided Verification (CAV)*, volume 3576 of *Lecture Notes in Computer Science*, pages 335–349, 2005.

[BCLZ04] Thomas Ball, Byron Cook, Shuvendu K. Lahiri, and Lintao Zhang. Zapato: Automatic theorem proving for predicate abstraction refinement. In R. Alur and D. Peled, editors, *Proceedings of the 16th International Conference on Computer Aided Verification, CAV'04 (Boston, Massachusetts)*, volume 3114 of *Lecture Notes in Computer Science*, pages 457–461. Springer, 2004.

[BD94] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *Procs. 6th Int. Conf. Computer Aided Verification (CAV)*, LNCS 818, pages 68–80, 1994.

[BDL96] C. Barrett, D. L. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In *Procs. 1st Intl. Conference on Formal Methods in Computer Aided Design*, LNCS 1166, pages 187–201, 1996.

[BdMS05] C. Barrett, L. de Moura, and A. Stump. SMT-COMP: Satisfiability Modulo Theories Competition. In K. Etessami and S. Rajamani, editors, *17th International Conference on Computer Aided Verification*, Lecture Notes in Computer Science, pages 20–23. Springer, 2005. Results at: www.csl.sri.com/users/demoura/smt-comp.

[BDS02] Clarke Barrett, David Dill, and Aaron 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.

[BEGJ00] Maria Luisa Bonet, Juan Luis Esteban, Nicola Galesi, and Jan Johannsen. On the relative complexity of resolution refinements and cutting planes proof systems. *SIAM J. Comput.*, 30(5):1462–1484, 2000.

[BGV01] R. Bryant, S. German, and M. Velev. Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. *ACM Trans. Computational Logic*, 2(1):93–134, 2001.

[BKS03] Paul Beame, Henry Kautz, and Ashish Sabharwal. On the power of clause learning. In *Proceedings of IJCAI-03, 18th International Joint Conference on Artificial Intelligence*, Acapulco, MX, 2003.

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

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

[dMR02] Leonardo de Moura and Harald 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] Leonardo de Moura and Harald Ruess. An experimental evaluation of ground decision procedures. In R. Alur and D. Peled, editors, *Proceedings of the 16th International Conference on Computer Aided Verification,* *CAV'04 (Boston, Massachusetts)*, volume 3114 of *Lecture Notes in Computer Science*, pages 162–174. Springer, 2004.

[dMRS04] Leonardo de Moura, Harald Rueß, and Natarajan Shankar. Justifying equality. In *Proceedings of the Second Workshop on Pragmatics of Decision Procedures in Automated Reasoning*, Cork, Ireland, 2004.

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

[DST80] Peter J. Downey, Ravi Sethi, and Robert E. Tarjan. Variations on the common subexpressions problem. *J. of the Association for Computing Machinery*, 27(4):758–771, 1980.

[ES03] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 502–518, 2003.

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

[FORS01] J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: Integrated Canonization and Solving (Tool prese ntation). In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of CAV'2001*, volume 2102 of *Lecture Notes in Computer Science*, pages 246–249. Springer-Verlag, 2001.

[GHN+04] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): Fast Decision Procedures. In R. Alur and D. Peled, editors, *Proceedings of the 16th International Conference on Computer Aided Verification, CAV'04 (Boston, Massachusetts)*, volume 3114 of *Lecture Notes in Computer Science*, pages 175–188. Springer, 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] Shuvendu K. Lahiri and Sanjit A. Seshia. The uclid decision procedure. In *Computer Aided Verification, 16th International Conference, (CAV)*, volume 3114 of *Lecture Notes in Computer Science*, pages 475–478, 2004.

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

[MS05a] Panagiotis Manolios and Sudarshan K. Srinivasan. A computationally efficient method based on commitment refinement maps for verifying pipelined machines. In *ACM IEEE Int. Conf. on Formal Methods and Models for Co-Design (MEMOCODE)*, 2005.

[MS05b] Panagiotis Manolios and Sudarshan K. Srinivasan. Refinement maps for efficient verification of processor models. In *Design, Automation and Test in Europe Conference and Exposition (DATE)*, pages 1304–1309. IEEE Computer Society, 2005.

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

[NO79] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.

[NO03] Robert Nieuwenhuis and Albert Oliveras. Congruence Closure with Integer Offsets. In M Vardi and A Voronkov, editors, *10h Int. Conf. Logic for Programming, Artif. Intell. and Reasoning (LPAR)*, LNAI 2850, pages 78–90, 2003.

[NO05a] Robert Nieuwenhuis and Albert Oliveras. BarcelogicTools for SMT, July 2005. SMT Competition 2005. Entrants' system descriptions. www.csl.sri.com/users/demoura/smt-comp.

[NO05b] Robert Nieuwenhuis and Albert Oliveras. DPLL(T) with Exhaustive Theory Propagation and its Application to Difference Logic. In Kousha Etessami and Sriram K. Rajamani, editors, *Proceedings of the 17th International Conference on Computer Aided Verification, CAV'05 (Edimburgh, Scotland)*, volume 3576 of *Lecture Notes in Computer Science*, pages 321–334. Springer, July 2005.

[NO05c] Robert Nieuwenhuis and Albert Oliveras. Proof-Producing Congruence Closure. In Jürgen Giesl, editor, *Proceedings of the 16th International Conference on Term Rewriting and Applications, RTA'05 (Nara, Japan)*, volume 3467 of *Lecture Notes in Computer Science*, pages 453–468. Springer, June 2005.

[NOT05] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Abstract DPLL and Abstract DPLL Modulo Theories. In Franz Baader and Andrei Voronkov, editors, *"11th Int. Conf. Logic for Programming, Artif. Intell. and Reasoning (LPAR)"*, volume 3452 of *Lecture Notes in Computer Science*, pages 36–50. Springer, 2005.

[RT03] Silvio Ranise and Cesare Tinelli. The SMT-LIB Format: An Initial Proposal. In *Proceedings of the 1st Workshop on Pragmatics of Decision Procedures in Automated Reasoning*, Miami, 2003.

[Rya04] Lawrence Ryan. Efficient Algorithms for Clause-Learning SAT Solvers. Master's thesis, School of Computing Science, Simon Fraser University, 2004.

[Sho84] Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, January 1984.

[ST05] Aaron Stump and Li-Yang Tan. The algebra of equality proofs. In Jürgen Giesl, editor, *Proceedings of the 16th International Conference on Term Rewriting and Applications, RTA'05 (Nara, Japan)*, volume 3467 of *Lecture Notes in Computer Science*, pages 469–483. Springer, 2005.

[Str02] Ofer Strichman. On solving presburger and linear arithmetic with sat. In Mark Aagaard and John W. O'Leary, editors, *Formal Methods in Computer-Aided Design, 4th International Conference, FMCAD 2002, Portland, OR, USA, November 6-8, 2002, Proceedings*, volume 2517 of *Lecture Notes in Computer Science*, pages 160–170. Springer, 2002.

[ZMMM01] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *Int. Conf. on Computer-Aided Design (ICCAD'01)*, pages 279–285, 2001.