Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL(T)

ROBERT NIEUWENHUIS AND ALBERT OLIVERAS

Technical University of Catalonia, Barcelona, Spain

AND

CESARE TINELLI

The University of Iowa, Iowa City, Iowa

Abstract. We first introduce *Abstract DPLL*, a rule-based formulation of the Davis–Putnam–Logemann–Loveland (DPLL) procedure for propositional satisfiability. This abstract framework allows one to cleanly express practical DPLL algorithms and to formally reason about them in a simple way. Its properties, such as soundness, completeness or termination, immediately carry over to the modern DPLL implementations with features such as backjumping or clause learning.

We then extend the framework to Satisfiability Modulo background Theories (SMT) and use it to model several variants of the so-called *lazy approach* for SMT. In particular, we use it to introduce a few variants of a new, efficient and modular approach for SMT based on a general DPLL(X) engine, whose parameter X can be instantiated with a specialized solver Solver_T for a given theory T, thus producing a DPLL(T) system. We describe the high-level design of DPLL(X) and its cooperation with Solver_T, discuss the role of *theory propagation*, and describe different DPLL(T) strategies for some theories arising in industrial applications.

Our extensive experimental evidence, summarized in this article, shows that DPLL(T) systems can significantly outperform the other state-of-the-art tools, frequently even in orders of magnitude, and have better scaling properties.

Categories and Subject Descriptors: B.6.3 [**Logic Design**]: Design Aids—*Verification*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*Computational logic*; *verification*; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving—*Deduction (e.g., natural, rule-based)*

General Terms: Theory, Verification

Additional Key Words and Phrases: SAT solvers, Satisfiability Modulo Theories

This work was partially supported by Spanish Ministry of Education and Science through the Logic Tools project TIN2004-03382 (all authors), the FPU grant AP2002-3533 (Oliveras) and National Science Foundation (NSF) grant 0237422 (Tinelli and Oliveras).

Authors' addresses: R. Nieuwenhuis and A. Oliveras, Technical University of Catalonia, Campus Nord—Edif. Omega, C. Jordi Girona, 1–3, 08034 Barcelona, Spain; C. Tinelli, University of Iowa, Department of Computer Science, 14 MacLean Hall, Iowa City, IA, 52242.

Journal of the ACM, Vol. 53, No. 6, November 2006, pp. 937–977.

1. *Introduction*

The problem of deciding the satisfiability of propositional formulas (SAT) does not only lie at the heart of the most important open problem in complexity theory (P vs. NP), it is also at the basis of many practical applications in such areas as Electronic Design Automation, Verification, Artificial Intelligence, and Operations Research. Thanks to recent advances in SAT-solving technology, propositional solvers are becoming the tool of choice for attacking more and more practical problems.

Most state-of-the-art SAT solvers [Moskewicz et al. 2001; Goldberg and Novikov 2002; Eén and Sörensson 2003; Ryan 2004] today are based on different variations of the Davis-Putnam-Logemann-Loveland (DPLL) procedure [Davis and Putnam 1960; Davis et al. 1962]. Starting essentially with the work on the GRASP, SATO and Relsat systems [Marques-Silva and Sakallah 1999; Zhang 1997; Bayardo and Schrag 1997], the spectacular improvements in the performance of DPLL-based SAT solvers achieved in the last years are due to (i) better implementation techniques, such as the *two-watched literal* approach for unit propagation, and (ii) several conceptual enhancements on the original DPLL procedure, aimed at reducing the amount of explored search space, such as *backjumping* (a form of *non-chronological backtracking*), *conflict-driven lemma learning*, and *restarts*. These advances make it now 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 to handle satisfiability problems in more expressive logics than propositional logic. In particular, they have been used to build efficient algorithms for the *Satisfiability Modulo Theories (SMT)* problem: deciding the satisfiability of ground first-order formulas with respect to background theories such as the theory of equality, of the integer or real numbers, of arrays, and so on [Armando et al. 2000, 2004; Filliâtre et al. 2001; Barrett et al. 2002; de Moura and Rueß 2002; Flanagan et al. 2003; Ganzinger et al. 2004; Bozzano et al. 2005]. SMT problems arise in many industrial applications, especially in formal verification (see Section 3 for examples). They may contain thousands of clauses like $p \vee \neg q \vee a = f(b - c) \vee g(g(b)) \neq c \vee a - c \leq 7$, with purely propositional atoms as well as atoms over (combined) theories, such as the theory of the integers, or of Equality with Uninterpreted Functions (EUF).

Altogether, many variants and extensions of the DPLL procedure exist today. They are typically described in the literature informally and with the aid of pseudocode fragments. Therefore, it has become difficult for the newcomer to understand the precise nature of all these procedures, and for the expert to formally reason about their properties.

The first main contribution of this article is to address these shortcomings by providing *Abstract DPLL*, a uniform, declarative framework for describing DPLL-based solvers, both for propositional satisfiability and for satisfiability modulo theories. The framework allows one to describe the essence of various prominent approaches and techniques in terms of simple transition rules and rule application strategies. By abstracting away heuristics and implementation issues, it facilitates the understanding of DPLL at a conceptual level as well as its correctness and termination. For DPLL-based SMT approaches, it moreover provides a clean formulation and a basis for comparison of the different approaches.

The second main contribution of this article is a new modular architecture for building SMT solvers in practice, called DPLL(T), and a careful study of *theory propagation*, a refinement of SMT methods that can have a crucial impact on their performance.

The architecture is based on a general DPLL(X) engine, whose parameter X can be instantiated with a specialized solver $Solver_T$ for a given theory T, thus producing a system DPLL(T). Such systems can be implemented extremely efficiently and have good scaling properties: our Barcelogic implementation of DPLL(T) won four divisions at the 2005 SMT Competition [Barrett et al. 2005] (for the other three existing divisions it had no $Solver_T$ yet). The insights provided by our Abstract DPLL framework were an important factor in the success of our DPLL(T) architecture and its Barcelogic implementation. For instance, the abstract framework helped us in understanding the interactions between the DPLL(X) engine and the solvers, especially concerning the different forms of theory propagation, as well as in defining a good interface between both.

Section 2 of this article presents the propositional version of Abstract DPLL. It models DPLL procedures by means of simple *transition systems*. While abstract and declarative in nature, these transition systems can explicitly model the salient conceptual features of state-of-the-art DPLL-based SAT solvers, thus bridging the gap between logic-based calculi for DPLL and actual implementations. Within the Abstract DPLL formalism, we discuss in a clean and uniform way properties such as soundness, completeness, and termination. These properties immediately carry over to modern DPLL implementations with features such as backjumping and learning.

For backjumping systems, for instance, we achieve this by modeling backjumping by a general rule that encompasses several backtracking strategies—including basic chronological backtracking—and explaining how different systems implement the rule. Similarly, we model learning by general rules that show how devices such as conflict graphs are just one possibility for computing new lemmas. We also provide a general and simple termination argument for DPLL procedures that does not depend on an exhaustive enumeration of truth assignments; instead, it relies on a notion of search progress neatly expressing that search advances with the deduction of new unit clauses—the higher up in the search tree the better—which is the very essence of backjumping.

In Section 3, we go beyond propositional satisfiability, and extend the framework to *Abstract DPLL Modulo Theories*. As in the purely propositional case, this again allows us to express—and formally reason about—a number of current DPLL-based techniques for SMT, such as the various variants of the so-called *lazy approach* [Armando et al. 2000, 2004; Filliâtre et al. 2001; Audemard et al. 2002; Barrett et al. 2002; de Moura and Rueß 2002; Flanagan et al. 2003; Ball et al. 2004].

In Section 4, based on the Abstract DPLL Modulo Theories framework, we introduce our DPLL(T) approach for building SMT systems. We first describe two variants of DPLL(T), depending on whether theory propagation is done exhaustively or not. Once the DPLL(X) engine has been implemented, this approach becomes extremely flexible: a DPLL(T) system for a theory T is obtained by simply plugging in the corresponding theory solver $Solver_T$, which must only be able to deal with *conjunctions* of theory literals and conform to a minimal and simple set of additional requirements. We discuss the design of DPLL(X) and describe how DPLL(X) and $Solver_T$ cooperate. We also show that practical T-solvers can be designed to include theory propagation in an efficient way. A nontrivial issue is how to deal with conflict analysis and clause learning adequately in the context of theory propagation. Different options and possible problems for doing this are analyzed and discussed in detail in Section 5.

In Section 6, we discuss some experiments with our Barcelogic implementation of DPLL(T). The results show that it can significantly outperform the best state-of-the-art tools and, in addition, scales up very well.

This article consolidates and improves upon preliminary ideas and results presented at the JELIA [Tinelli 2002], LPAR [Nieuwenhuis and Oliveras 2003; Nieuwenhuis et al. 2005], and CAV [Ganzinger et al. 2004; Nieuwenhuis and Oliveras 2005a] conferences.

2. *Abstract DPLL in the Propositional Case*

We start this section with some formal preliminaries on propositional logic and on transition systems. Then we introduce several variants of Abstract DPLL and prove their correctness properties, showing at the same time how the different features of actual DPLL implementations are modeled by these variants.

2.1. Formulas, Assignments, and Satisfaction.

Let P be a fixed finite set of propositional symbols. If $p \in P$, then p is an *atom* and 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 $l_1 \vee \cdots \vee l_n$. A *unit clause* is a clause consisting of a single literal. A (CNF) *formula* is a conjunction of one or more clauses $C_1 \wedge \cdots \wedge C_n$. When it leads to no ambiguities, we will sometimes also write such a formula in set notation $\{C_1, \ldots, C_n\}$, or simply replace the $\wedge$ 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 $l \in M$, is *false* in M if $\neg l \in M$, and is *undefined* in M otherwise. A literal is *defined* in M if it is either true or false in M. The assignment 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 a *model* of F. If F has no models then it is *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 denote assignments. If C is a clause $l_1 \vee \cdots \vee l_n$, we sometimes write $\neg C$ to denote the formula $\neg l_1 \wedge \cdots \wedge \neg l_n$.

2.2. States and Transition Systems in Abstract DPLL.

DPLL can be fully described by simply considering that a *state* of the procedure is either the distinguished state *FailState* or a pair of the form $M \parallel F$, where F is a CNF formula, that is, a finite set of clauses, and M is, essentially, a (partial) assignment.

More precisely, M is a *sequence* of literals, never containing both a literal and its negation, where each literal has an *annotation*, a bit that marks it as a *decision* literal (see below) or not. Frequently, we will consider M just as a partial assignment, or as a set or conjunction of literals (and hence as a formula), ignoring both the annotations and the order between its elements.

The concatenation of two such sequences will be denoted by simple juxtaposition. When we want to emphasize that a literal l is annotated as a decision literal we will write it as l^{d}. We will denote the empty sequence of literals (or the empty assignment) by $\emptyset$. We say that a clause C is *conflicting* in a state $M \parallel F, C$ if $M \models \neg C$.

We will model each DPLL procedure by means of a set of states together with a binary relation $\Longrightarrow$ over these states, called the *transition relation*. As usual, we use infix notation, writing $S \Longrightarrow S'$ instead of $(S, S') \in \Longrightarrow$. If $S \Longrightarrow S'$ we say that there is a *transition* from S to S'. We denote by $\Longrightarrow^*$ the reflexive-transitive closure of $\Longrightarrow$. We call any sequence of transitions of the form $S_0 \Longrightarrow S_1,\ \ S_1 \Longrightarrow S_2, \cdots$ a *derivation*, and denote it by $S_0 \Longrightarrow S_1 \Longrightarrow S_2 \Longrightarrow \cdots$. We call any subsequence of a derivation a *subderivation*.

In what follows, transition relations will be defined by means of conditional *transition rules*. For a given state S, a transition rule precisely defines whether there is a transition from S by this rule and, if so, to which state S'. Such a transition is called an *application step* of the rule.

A *transition system* is a set of transition rules defined over some given set of states. Given a transition system R, the transition relation defined by R will be denoted by $\Longrightarrow_R$. If there is no transition from S by $\Longrightarrow_R$, we will say that S is *final* with respect to R (examples of a transition system and a final state with respect to it can be found in Definition 2.1 and Example 2.2).

2.3. THE CLASSICAL DPLL PROCEDURE. A very simple DPLL system, faithful to the classical DPLL algorithm, consists of the following five transition rules. We give this system here mainly for explanatory and historical reasons. The informally stated results for it are easily obtained by adapting the more general ones given in Section 2.5.

Definition 2.1. The *Classical DPLL system* is the transition system Cl consisting of the following five transition rules. In this system, the literals added to M by all rules except Decide are annotated as non-decision literals.

UnitPropagate :

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

PureLiteral :

$$M \parallel F \quad \Longrightarrow \quad M\ l \parallel F \quad \textbf{if} \begin{cases} l \text{ occurs in some clause of } F \\ \neg l \text{ occurs in no clause of } F \\ l \text{ is undefined in } M. \end{cases}$$

Decide :

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

Fail :

$$M \parallel F,\ C \quad \Longrightarrow \quad \mathit{FailState} \quad \textbf{if} \begin{cases} M \models \neg C \\ M \text{ contains no decision literals.} \end{cases}$$

Backtrack :

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

One can use the transition system Cl for deciding the satisfiability of an input formula F simply by generating an arbitrary derivation $\emptyset \parallel F \Longrightarrow_{\mathrm{Cl}} \cdots \Longrightarrow_{\mathrm{Cl}} S_n$, where S_n is a final state with respect to Cl. The applicability of each of the five rules is easy to check and, as we will see, their application always leads to finite derivations. Moreover, for every derivation like the above ending in a final state S_n, (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. Note that in this Classical DPLL system the second component of a state remains unchanged, a property that does not hold for the other transition systems we introduce later.

We now briefly comment on what the different rules do. In the following, if M is a sequence of the form $M_0\ l_1\ M_1 \cdots l_k\ M_k$, where the l_i are all the decision literals in M, we say that the state $M \parallel F$ is *at decision level* k, and that all the literals of each $l_i\ M_i$ *belong to decision level* i.

—UnitPropagate: To satisfy a CNF formula, all its clauses have to be true. Hence, if a clause of F contains a literal l whose truth value is not defined by the current assignment M while all the remaining literals of the clause are false, then M must be extended to make l true.

—PureLiteral: If a literal l is *pure* in F, that is, it occurs in F while its negation does not, then F is satisfiable only if it has a model that makes l true. Thus, if M does not define l it can be extended to make l true.

—Decide: This 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 the alternative extension $M\ \neg l$ must still be considered. This is done by means of the Backtrack rule.

—Fail: This rule detects a *conflicting clause* C and produces the *FailState* state whenever M contains no decision literals.

—Backtrack: If a conflicting clause C is detected and Fail does not apply, then the rule backtracks one *decision level*, by replacing the most recent decision literal l^{d} by $\neg l$ and removing any subsequent literals in the current assignment. Note that $\neg l$ is annotated as a nondecision literal, since the other possibility l has already been explored.

Example 2.2. The following is a derivation in the Classical DPLL system, with each transition annotated by the rule that makes it possible. To improve readability we denote atoms by natural numbers, and negation by overlining.

$$\begin{array}{rcllllll}
\emptyset & \parallel & \overline{1}\lor\overline{2}, & 2\lor 3, & \overline{1}\lor\overline{3}\lor 4, & 2\lor\overline{3}\lor\overline{4}, & 1\lor 4 & \Longrightarrow_{\mathrm{Cl}}\ \text{(Decide)} \\
1^{\mathsf{d}} & \parallel & \overline{1}\lor\overline{2}, & 2\lor 3, & \overline{1}\lor\overline{3}\lor 4, & 2\lor\overline{3}\lor\overline{4}, & 1\lor 4 & \Longrightarrow_{\mathrm{Cl}}\ \text{(UnitPropagate)} \\
1^{\mathsf{d}}\ \overline{2} & \parallel & \overline{1}\lor\overline{2}, & 2\lor 3, & \overline{1}\lor\overline{3}\lor 4, & 2\lor\overline{3}\lor\overline{4}, & 1\lor 4 & \Longrightarrow_{\mathrm{Cl}}\ \text{(UnitPropagate)} \\
1^{\mathsf{d}}\ \overline{2}\ 3 & \parallel & \overline{1}\lor\overline{2}, & 2\lor 3, & \overline{1}\lor\overline{3}\lor 4, & 2\lor\overline{3}\lor\overline{4}, & 1\lor 4 & \Longrightarrow_{\mathrm{Cl}}\ \text{(UnitPropagate)} \\
1^{\mathsf{d}}\ \overline{2}\ 3\ 4 & \parallel & \overline{1}\lor\overline{2}, & 2\lor 3, & \overline{1}\lor\overline{3}\lor 4, & 2\lor\overline{3}\lor\overline{4}, & 1\lor 4 & \Longrightarrow_{\mathrm{Cl}}\ \text{(Backtrack)} \\
\overline{1} & \parallel & \overline{1}\lor\overline{2}, & 2\lor 3, & \overline{1}\lor\overline{3}\lor 4, & 2\lor\overline{3}\lor\overline{4}, & 1\lor 4 & \Longrightarrow_{\mathrm{Cl}}\ \text{(UnitPropagate)} \\
\overline{1}\ 4 & \parallel & \overline{1}\lor\overline{2}, & 2\lor 3, & \overline{1}\lor\overline{3}\lor 4, & 2\lor\overline{3}\lor\overline{4}, & 1\lor 4 & \Longrightarrow_{\mathrm{Cl}}\ \text{(Decide)} \\
\overline{1}\ 4\ \overline{3}^{\mathsf{d}} & \parallel & \overline{1}\lor\overline{2}, & 2\lor 3, & \overline{1}\lor\overline{3}\lor 4, & 2\lor\overline{3}\lor\overline{4}, & 1\lor 4 & \Longrightarrow_{\mathrm{Cl}}\ \text{(UnitPropagate)} \\
\overline{1}\ 4\ \overline{3}^{\mathsf{d}}\ 2 & \parallel & \overline{1}\lor\overline{2}, & 2\lor 3, & \overline{1}\lor\overline{3}\lor 4, & 2\lor\overline{3}\lor\overline{4}, & 1\lor 4 &
\end{array}$$

The last state of this derivation is final. The (total) assignment in it is a model of the formula.

The Davis–Putnam procedure [Davis and Putnam 1960] 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 Davis et al. [1962], mostly focused on the propositional phase. What most authors now 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.

2.4. 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—hence, we will not consider this rule in the following. Moreover, *backjumping*, a more general and more powerful backtracking mechanism, is now commonly used in place of chronological backtracking.

The usefulness of a more sophisticated backtracking mechanism for DPLL solvers is perhaps best illustrated with another example of derivation in the Classical DPLL system.

Example 2.3.

$$\begin{array}{rcll}
\emptyset & \| & \overline{1}\vee 2,\ \ \overline{3}\vee 4,\ \ \overline{5}\vee\overline{6},\ \ 6\vee\overline{5}\vee\overline{2} & \Longrightarrow_{\mathrm{B}}\ \textsf{(Decide)}\\
1^{\mathsf{d}} & \| & \overline{1}\vee 2,\ \ \overline{3}\vee 4,\ \ \overline{5}\vee\overline{6},\ \ 6\vee\overline{5}\vee\overline{2} & \Longrightarrow_{\mathrm{B}}\ \textsf{(UnitPropagate)}\\
1^{\mathsf{d}}\ 2 & \| & \overline{1}\vee 2,\ \ \overline{3}\vee 4,\ \ \overline{5}\vee\overline{6},\ \ 6\vee\overline{5}\vee\overline{2} & \Longrightarrow_{\mathrm{B}}\ \textsf{(Decide)}\\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}} & \| & \overline{1}\vee 2,\ \ \overline{3}\vee 4,\ \ \overline{5}\vee\overline{6},\ \ 6\vee\overline{5}\vee\overline{2} & \Longrightarrow_{\mathrm{B}}\ \textsf{(UnitPropagate)}\\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}}\ 4 & \| & \overline{1}\vee 2,\ \ \overline{3}\vee 4,\ \ \overline{5}\vee\overline{6},\ \ 6\vee\overline{5}\vee\overline{2} & \Longrightarrow_{\mathrm{B}}\ \textsf{(Decide)}\\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}}\ 4\ 5^{\mathsf{d}} & \| & \overline{1}\vee 2,\ \ \overline{3}\vee 4,\ \ \overline{5}\vee\overline{6},\ \ 6\vee\overline{5}\vee\overline{2} & \Longrightarrow_{\mathrm{B}}\ \textsf{(UnitPropagate)}\\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}}\ 4\ 5^{\mathsf{d}}\ \overline{6} & \| & \overline{1}\vee 2,\ \ \overline{3}\vee 4,\ \ \overline{5}\vee\overline{6},\ \ 6\vee\overline{5}\vee\overline{2} & \Longrightarrow_{\mathrm{B}}\ \textsf{(Backtrack)}\\
1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}}\ 4\ \overline{5} & \| & \overline{1}\vee 2,\ \ \overline{3}\vee 4,\ \ \overline{5}\vee\overline{6},\ \ 6\vee\overline{5}\vee\overline{2} &
\end{array}$$

Before the Backtrack step, the clause $6 \vee \overline{5} \vee \overline{2}$ is conflicting: it is false in the assignment $1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}}\ 4\ 5^{\mathsf{d}}\ \overline{6}$. This is a consequence of the unit propagation 2 of the decision 1^{d}, together with the decision 5^{d} and its unit propagation $\overline{6}$.

Therefore, one can infer that the decision 1^{d} is incompatible with the decision 5^{d}, that is, that the given clause set entails $\overline{1}\vee\overline{5}$. Similarly, it also entails $\overline{2}\vee\overline{5}$.

Such entailed clauses are called *backjump clauses* if their presence would have allowed a unit propagation at an earlier decision level. This is precisely what *backjumping* does: given a backjump clause, it goes back to that level and adds the unit propagated literal. For example, using $\overline{2}\vee\overline{5}$ as a backjump clause, the last Backtrack step could be replaced by a backjump to a state with first component $1^{\mathsf{d}}\ 2\ \overline{5}$.

We model all this in the next system with the Backjump rule, of which Backtrack is a particular case. In this rule, the clause $C' \vee l'$ is the backjump clause, where l'

is the literal that can be unit propagated ($\overline{5}$ in our example). Below we show that the rule is effective: a backjump clause can always be found.

Definition 2.4. The *Basic DPLL system* is the four-rule transition system B consisting of the rules UnitPropagate, Decide, Fail from Classical DPLL, and the following Backjump rule:

Backjump :

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

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

Chronological backtracking, modeled by Backtrack, always undoes the *last* decision l, going back to the previous level and adding $\neg l$ to it. *Conflict-driven* backjumping, as modeled by Backjump, is generally able to backtrack further than chronological backtracking by analyzing the reasons that produced the conflicting clause. Backjump can frequently undo *several* decisions at once, going back to a lower decision level than the previous level and adding some new literal to that lower level. It jumps over levels that are irrelevant to the conflict. In the previous example, it jumps over the decision 3^{d} and its consequence 4, which are totally unrelated with the reasons for the falsity of the conflicting clause $6 \vee \overline{5} \vee \overline{2}$. Moreover, intuitively, the search state $1^{\mathsf{d}}\ 2\ \overline{5}$ reached after Backjump is more *advanced* than the state $1^{\mathsf{d}}\ 2\ 3^{\mathsf{d}}\ 4\ \overline{5}$ reached after Backtrack. This notion of "being more advanced" is formalized in Theorem 2.10 below.

We show in the proof of Lemma 2.8 below that the literals of the backjump clause can always be chosen among the negations of the decision literals—although better choices usually exist. When the negations of *all* the decision literals are included in the backjump clause, the Backjump rule simulates the Backtrack rule of Classical DPLL. We remark that, in fact, Lemma 2.8 shows that, whenever a state $M \parallel F$ contains a conflicting clause, either Fail applies, if there are no decision literals in M, or otherwise Backjump applies.

Most modern DPLL implementations make additional use of backjump clauses: they add them to the clause set as *learned* clauses, also called *lemmas*, implementing what is usually called *conflict-driven learning*.

In Example 2.3, learning the clause $\overline{2} \vee \overline{5}$ will allow the application of UnitPropagate to any state whose assignment contains either 2 or 5. Hence, it will prevent any conflict caused by having both 2 and 5 in M. Reaching such *similar* conflicts frequently happens in industrial problems having some regular structure, and learning such lemmas has been shown to be very effective in improving performance.

Since a lemma is aimed at preventing future similar conflicts, when these conflicts are not very likely to be found again the lemma can be removed. In practice, a lemma is removed when its *relevance* (see, e.g., Bayardo and Schrag [1997]) or its *activity* level drops below a certain threshold; the activity can be, for example, the number of times it becomes a unit or a conflicting clause [Goldberg and Novikov 2002].

To model lemma learning and removal we consider the following extension of the Basic DPLL system.

Definition 2.5. The *DPLL system with learning*, denoted by L, consists of the four transition rules of the Basic DPLL system and the two additional rules:

$$\begin{array}{llll}
\mathsf{Learn}: & & & \\
\quad M \parallel F & \Longrightarrow M \parallel F,\, C & \textbf{if} & \begin{cases} \text{each atom of } C \text{ occurs in } F \text{ or in } M \\ F \models C. \end{cases} \\
\mathsf{Forget}: & & & \\
\quad M \parallel F,\, C & \Longrightarrow M \parallel F & \textbf{if} & \big\{\, F \models C.
\end{array}$$

In any application step of Learn, the clause C is said to be *learned* if it did not already belong to F. Similarly, it is said to be *forgotten* by Forget.

Observe that the Learn rule allows one to add to the current formula F an arbitrary clause C entailed by F, as long as all the atoms of C occur in F or M. This models not only conflict-driven lemma learning but also any other techniques that produce consequences of F, such as limited forms of resolution (see the following example).

Similarly, the Forget rule can be used in principle to remove from F *any* clause that is entailed by the rest of F, not just those previously added to the clause set by Learn. The applicability of the two rules in their full scope, however, is limited in practice by the relative cost of determining such entailments in general.

The six rules of the DPLL system with learning model the high-level conceptual structure of DPLL implementations. These rules will allow us to formally reason about properties such as correctness or termination.

Example 2.6. We now show how the Backjump rule can be guided by means of a *conflict graph* for finding backjump clauses. In this example we assume a strategy that is followed in most SAT solvers: (i) Decide is applied only if no other Basic DPLL rule is applicable (Theorem 5.2 of Section 5 shows that this is not needed, but here we require it for simplicity) and (ii) after each application of Backjump, the backjump clause is learned.

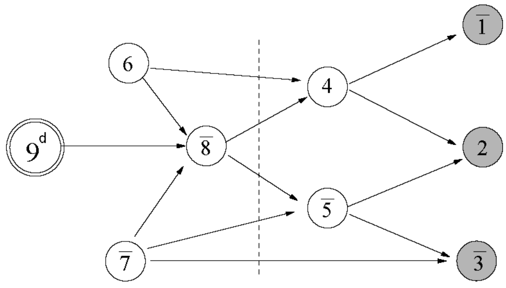
Consider a state of the form $M \parallel F$ where, among other clauses, F contains:

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

and M is of the form: $\ldots 6 \ldots \overline{7} \ldots 9^{\mathsf{d}}\ \overline{8}\ \overline{5}\ 4\ \overline{1}\ 2\ \overline{3}$.

It is easy to see that this state can be reached after the last decision 9^{d} by six applications of UnitPropagate. For example, $\overline{8}$ is implied by 9, 6, and $\overline{7}$ because of the clause $\overline{9}\vee\overline{6}\vee 7\vee\overline{8}$. A typical DPLL implementation will save the sequence of propagated literals and remember for each one of them the clause that caused its propagation. Now, in the state $M \parallel F$ above the clause $1\vee\overline{2}\vee 3$ is conflicting, since M contains $\overline{1}$, 2 and $\overline{3}$. Using the saved information, the DPLL implementation can trace back the reasons for this conflicting clause. For example, the saved data will show that $\overline{3}$ was implied by $\overline{5}$ and $\overline{7}$, due to the clause $5\vee 7\vee\overline{3}$. The literal $\overline{5}$ was in turn implied by $\overline{8}$ and $\overline{7}$, and so on.

This way, *working backwards from the conflicting clause* and in the opposite order in which each literal was propagated, it is possible to build the following *conflict graph*, where the nodes corresponding to the conflicting clause are shown in gray:



This figure shows the graph obtained when the decision literal of the current decision level (here, 9^{d}) is reached in this backwards process—which is why this node and the nodes belonging to earlier decision levels (in this example, literals 6 and $\overline{7}$) have no incoming arrows.

To find a backjump clause, it suffices to cut the graph into two parts. The first part must contain (at least) all the literals with no incoming arrows. The second part must contain (at least) all the literals with no outgoing arrows, that is, 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 in the graph, where the literals with cut outgoing edges are $\overline{8}$, $\overline{7}$, and 6. 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, that is, one can infer the clause $8 \vee 7 \vee \overline{6}$. In this case, this is a possible backjump clause, that is, the clause $C' \vee l'$ in the definition of the Backjump rule, with the literal 8 playing the role of l'. The clause allows one to backjump to the decision level of $\overline{7}$ and add 8 to it. After that, the clause $8 \vee 7 \vee \overline{6}$ has to be learned to explain in future conflicts the presence of 8 as a propagation from 6 and $\overline{7}$.

The kind of cuts we have described produce backjump clauses provided that exactly one of the literals with cut outgoing edges belongs to the current decision level. The negation of this literal will act as the literal l' in the backjump rule. In the SAT literature, the literal is called a *Unique Implication Point (UIP)* of the conflict graph. Formally, UIPs are defined as follows. Let D be the set of all the literals of a conflicting clause C that have become false at the current decision level (this set is always nonempty, since Decide is applied only if Fail or Backjump do not apply). A UIP in the conflict graph of C is any literal that belongs to all paths in the graph from the current decision literal to the negation of a literal in D. Note that a conflict graph always contains at least one UIP, the decision literal itself, but in general it can contain more (in our example 9^{d} and $\overline{8}$ are both UIPs).

In practice, it is not actually necessary to build the conflict graph to produce a backjump clause; it suffices to work backwards from the conflicting clause, maintaining only a *frontier* list of literals yet to be expanded, until the *first* UIP (first in the reverse propagation ordering) has been reached [Marques-Silva and Sakallah 1999; Zhang et al. 2001].

The construction of the backjump clause can also be seen as a derivation in the resolution calculus, constructed according to the following *backwards conflict resolution* process. In our example, the clause $8 \vee 7 \vee \overline{6}$ is obtained by successive resolution steps on the conflicting clause, resolving away the literals 3, $\overline{2}$, 1, $\overline{4}$ and 5, in the reverse order their negations were propagated, with the respective clauses that caused the propagations:

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

The process stops once it generates a clause with only one literal of the current decision level, which is precisely the first UIP (in our example, the literal 8 in the clause $8 \vee 7 \vee \overline{6}$). Some SAT solvers, such as Siege, also learn some of the intermediate clauses in such resolution derivations [Ryan 2004].

2.5. CORRECTNESS OF DPLL WITH LEARNING. In this section, we show how the DPLL system with learning can be used as a decision procedure for the satisfiability of CNF formulas.

Deciding the satisfiability of an input formula F will be done by generating an arbitrary derivation of the form $\emptyset \parallel F \Longrightarrow_{\mathrm{L}} \cdots \Longrightarrow_{\mathrm{L}} S_n$ such that S_n is *final with respect to the Basic DPLL system*. Note that final states with respect to the DPLL system with Learning do not always exist, since the same clause could be learned and forgotten infinitely many times.

For all rules their applicability is easy to check and, as we will show in Theorem 2.11, if infinite subderivations with only Learn and Forget steps are avoided, one always reaches a state that is final with respect to the Basic DPLL system. This state S is moreover easily recognizable as final, because it is either *FailState* or of form $M \parallel F'$ where F' has no conflicting clauses and all of its literals are defined in M. Furthermore, similarly to the Classical DPLL system and as proved below in Theorem 2.12; in the first case, F is unsatisfiable, in the second case, it is satisfied by M.

We emphasize that these formal results apply to *any* procedure modeled by the DPLL system with learning, and can moreover be extended to DPLL Modulo Theories. This generalizes the less formal correctness proof for the concrete pseudo code of the Chaff algorithm given in Zhang and Malik [2003], which has the same underlying proof idea.

The starting point for our results is the next lemma, which lists a few properties that are invariant for all the states derived in the DPLL system with learning from initial states of the form $\emptyset \parallel F$.

LEMMA 2.7. *If* $\emptyset \parallel F \Longrightarrow_{\mathrm{L}}{}^{*} M \parallel G$, *then the following hold.*

(1) *All the atoms in* M *and all the atoms in* G *are atoms of* F.

(2) M *contains no literal more than once and is indeed an assignment, that is, it contains no pair of literals of the form* p *and* $\neg p$.

(3) G *is logically equivalent to* F.

(4) *If M is of the form $M_0\ l_1\ M_1 \cdots l_n\ M_n$, where $l_1, \ldots, l_n$ are all the decision literals of M, then $F, l_1, \ldots, l_i \models M_i$ for all i in $0 \ldots n$.*

PROOF. Since all four properties trivially hold in the initial state $\emptyset \parallel F$, we only need to prove that all six rules preserve them. Consider a step $M' \parallel F' \Longrightarrow_{\mathrm{L}} M'' \parallel F''$ and assume all properties hold in $M' \parallel F'$. Property 1 holds in $M'' \parallel F''$ because the only atoms that may be added to M'' or F'' are the ones in F' or M', all of which belong to F. The side conditions of the rules clearly preserve Property 2. As for Property 3, only Learn and Forget may break the invariant. But learning (or forgetting) a clause C that is a logical consequence clearly preserves equivalence between F' and F''.

For the fourth property, consider that M' is of the form $M'_0\ l_1\ M'_1 \cdots l_n\ M'_n$, and $l_1, \ldots, l_n$ are all the decision literals of M'. If the step is an application of Decide, there is nothing to prove. For Learn or Forget, it easily follows since M' is M'' and F'' is logically equivalent to F'. The remaining rules are:

UnitPropagate: Since M'' will be of the form $M'l$ (we use l and C as in the definition of the rule), we only have to prove that $F, l_1, \ldots, l_n \models l$, which holds since (i) $F, l_1, \ldots, l_n \models M'$, (ii) $M' \models \neg C$, (iii) $C \vee l$ is a clause of F' and (iv) F and F' are equivalent.

Backjump: Assume that, in the Backjump rule, l^{d} is l_{j+1}, the $j+1$-th decision literal. Then (using l' and C' as in the definition of the rule), M'' is of the form $M'_0\ l_1\ M'_1 \cdots l_j\ M'_j\ l'$. We only need to show that $F, l_1, \ldots, l_j \models l'$. This holds as for the UnitPropagate case, since we have (i) $F, l_1, \ldots, l_j \models M'_0\ l_1\ M'_1 \cdots l_j\ M'_j$, (ii) $M'_0\ l_1\ M'_1 \cdots l_j\ M'_j \models \neg C'$, (iii) $F' \models C' \vee l'$ and (iv) F and F' are equivalent. □

The most interesting property of this lemma is probably Property 4. It shows that every nondecision literal added to an assignment M is a logical consequence of the previous decision literals of M and the initial formula F. In other words, we have that $F, l_1, \ldots, l_n \models M$. Hence, the only arbitrary additions to M are the ones made by Decide.

Another important property concerns the applicability of Backjump. Given a state with a conflicting clause, it may not be clear a priori whether Backjump is applicable or not, mainly due to the need to find an appropriate backjump clause. Below we show that, if there is a conflicting clause, it is always the case that either Backjump or Fail applies. Moreover, whenever the first precondition of Backjump holds ($M\ l^{\mathrm{d}}\ N \models \neg C$), a backjump clause $C' \vee l'$ always exists and can be easily computed.

LEMMA 2.8. *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 $M \parallel F'$.*

PROOF. If there is no decision literal in M, it is immediate that Fail applies. Otherwise, M is of the form $M_0\ l_1\ M_1 \cdots l_n\ M_n$ for some $n > 0$, where $l_1, \ldots, l_n$ are all the decision literals of M. Since $M \models \neg C$ we have, due to Lemma 2.7-4, that $F, l_1, \ldots, l_n \models \neg C$. If we now consider any i in $1 \cdots n$ such that $F, l_1, \ldots, l_i \models \neg C$, and any j in $0 \cdots i-1$ such that $F, l_1, \ldots, l_j, l_i \models \neg C$, we can show that then backjumping to decision level j is possible.

Let C' be the clause $\neg l_1 \vee \cdots \vee \neg l_j$, and note that M is also of the form $M'\ l_{j+1}\ N$. Then Backjump is applicable to $M \parallel F'$, yielding the state $M'\ \neg l_i \parallel F'$. That is because the clause $C' \vee \neg l_i$ satisfies all the side conditions of the Backjump rule:

(i) $F' \models C' \vee \neg l_i$ because $F, l_1, \ldots, l_j, l_i \models \neg C$, which implies, given that C is in F' and F' is equivalent to F (by Lemma 2.7-3), that $F, l_1, \ldots, l_j, l_i$ is unsatisfiable or, equivalently, that $F \models \neg l_1 \vee \cdots \vee \neg l_j \vee \neg l_i$; furthermore, $M' \models \neg C'$ by construction of C';

(ii) $\neg l_i$ is undefined in M' (by Lemma 2.7-2);

(iii) l_i occurs in M. □

It is interesting to observe that, the smaller one can choose the value j in the previous proof, the higher one can backjump. Note also that, if we construct the backjump clause as in the proof and take i to be n and j to be $n-1$ then the Backjump rule models standard backtracking.

We stress that backjump clauses need not be built as in the proof above, out of the decision literals of the current assignment. It follows from the termination and correctness results given in this section that in practice one is free to apply the backjump rule with *any* backjump clause. In fact, backjump clauses may be built to contain no decision literals at all, as is for instance possible in backjumping SAT solvers relying on the first UIP learning scheme illustrated in Example 2.6.

Given the previous lemma, it is easy to prove that final states with respect to Basic DPLL will be either *FailState* or $M \parallel F'$, where M is a model of the original formula F. More formally:

LEMMA 2.9. *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*

(1) *all literals of* F' *are defined in* M,

(2) *there is no clause* C *in* F' *such that* $M \models \neg C$, *and*

(3) M *is a model of* F.

PROOF. Assume S is not *FailState*. If (1) does not hold, then S cannot be final, since Decide would be applicable. Similarly, for (2): by Lemma 2.8, either Fail or Backjump would apply. Together (1) and (2) imply that all clauses of F' are defined and true in M, and since by Lemma 2.7(3), F and F' are logically equivalent this implies that M is a model of F. □

We now prove termination of the Basic DPLL system.

THEOREM 2.10. *There are no infinite derivations of the form* $\emptyset \parallel F \Longrightarrow_{\mathrm{B}} S_1 \Longrightarrow_{\mathrm{B}} \cdots$.

PROOF. It suffices to define a well-founded strict partial ordering $\succ$ on states, and show that each step $M \parallel F \Longrightarrow_{\mathrm{B}} M' \parallel F$ is decreasing with respect to this ordering, that is, $M \parallel F \;\succ\; M' \parallel F$. Note that such an ordering must be entirely based on the first component of the states, because in this system without Learn and Forget the second component of states remains constant.

Let M be of the form $M_0\; l_1\; M_1 \cdots l_p\; M_p$, where $l_1, \ldots, l_p$ are all the decision literals of M. Similarly, let M' be $M'_0\; l'_1\; M'_1 \cdots l'_{p'}\; M'_{p'}$.

Let n be the number of distinct atoms (propositional variables) in F. By Lemma 2.7(1,2), we have that p, p' and the length of M and M' are always smaller than or equal to n.

For each assignment N, define $m(N)$ to be $n - \mathit{length}(N)$, that is, $m(N)$ is the number of literals "missing" in N for N to be total. Now define: $M \parallel F' \;\succ\; M' \parallel F''$ if

(i) there is some i with $0 \leq i \leq p, p'$ such that

$$m(M_0) = m(M'_0), \ldots, m(M_{i-1}) = m(M'_{i-1}), \quad m(M_i) > m(M'_i) \text{ or}$$

(ii) $m(M_0) = m(M'_0), \ldots, m(M_p) = m(M'_p)$ and $m(M) > m(M')$.

Note that, in case (ii), we have $p' > p$, and all decision levels up to p coincide in number of literals. Comparing the number of missing literals in sequences is clearly a strict ordering (i.e., it is an irreflexive and transitive relation) and it is also well-founded, and hence this also holds for its lexicographic extension on tuples of sequences of bounded length. It is easy to see that all Basic DPLL rules are decreasing with respect to $\succ$ if *FailState* is added as an additional minimal element. The rules UnitPropagate and Backjump decrease by case (i) of the definition and Decide decreases by case (ii). □

It is nice to see in this proof that, in contrast to the classical, depth-first DPLL procedure, progress in backjumping DPLL procedures is not measured by the number of decision literals that have been *tried* with both truth values, but by the number of defined literals that are added to earlier decision levels. The Backjump rule makes progress in this sense by increasing by one the number of defined literals in the decision level it backjumps to. The lower this decision level is (i.e., the higher up in the depth-first search tree), the more progress is made with respect to $\succ$.

As an immediate consequence of this theorem, we obtain the termination of the DPLL system with learning if infinite subderivations with only Learn and Forget steps are avoided. The reason is that the other steps (the Basic DPLL ones) decrease the first components of the states with respect to the well-founded ordering, while the Learn and Forget steps do not modify that component.

THEOREM 2.11. *Every derivation $\emptyset \parallel F \Longrightarrow_{\mathrm{L}} S_1 \Longrightarrow_{\mathrm{L}} \cdots$ by the DPLL system with Learning is finite if it contains no infinite subderivations consisting of only Learn and Forget steps.*

Note that this condition is very weak and easily enforced. Learn is typically only applied together with Backjump in order to learn the corresponding backjump clause. The theorem entails that such a strategy eventually reaches a state where only Learn and/or Forget apply, that is, a state that is final with respect to the Basic DPLL system. As already mentioned, by Lemma 2.9, this state is moreover easily recognizable because it is *FailState* or else it has the form $M \parallel G$ with all literals of G defined in M and no conflicting clause.

Actually, we could have alternatively defined a state $M \parallel G$ to be final if M is a *partial* assignment satisfying all clauses of G, hence allowing some literals of G to remain undefined. Then the correctness argument would have been exactly the same but without the use of Lemma 2.9—which now is needed mostly to show that the current definition of a final state $M \parallel G$ is a sufficient condition for M to be a model of G. However, in typical DPLL implementations, checking each time whether a partial assignment is a model of the current formula G is more expensive, because of the necessary additional bookkeeping, than just extending a partial model of G to a total one, which can be done with no search. But note that things may be different in the SMT case (see a brief discussion at the end of Section 3), or when the goal is to enumerate *all* models (perhaps in some compact representation) of the initial formula F.

We are now ready to prove that DPLL with learning provides a decision procedure for the satisfiability of CNF formulas.

THEOREM 2.12. *If* $\emptyset \parallel F \Longrightarrow_{\mathrm{L}}^{*} S$ *where* S *is final with respect to 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 2.7(3), we have that $F \models C$. However, if $M \models \neg C$, by Lemma 2.7(4), 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 2.9(3), M is a model of F and hence F is satisfiable.

For Property 2, if S is $M \parallel F'$, then, again by Lemma 2.9(3), M is a model of F. □

Note that the previous theorem does not guarantee confluence in the sense of rewrite systems, say. With unsatisfiable formulas, the only possible final (with respect to Basic DPLL) state for a sequence is *FailState*. If, on the other hand, the formula is satisfiable, different states that are final with respect to Basic DPLL may be reachable. However, all of them will be of the form $M \parallel F'$, with M a model of the original formula.

Although Theorem 2.12 was given for the relation $\Longrightarrow_{\mathrm{L}}$, it also holds for $\Longrightarrow_{\mathrm{B}}$, since the existence of Learn or Forget is not required in the proof.

THEOREM 2.13. *If* $\emptyset \parallel F \Longrightarrow_{\mathrm{B}}^{*} S$ *where* S *is final with respect to 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.

2.6. ABOUT PRACTICAL IMPLEMENTATIONS AND RESTARTS.

State-of-the art SAT-solvers [Moskewicz et al. 2001; Goldberg and Novikov 2002; Eén and Sörensson 2003; Ryan 2004] essentially apply Abstract DPLL with Learning using efficient implementation techniques for UnitPropagate (such as the two-watched literal scheme for unit propagation [Moskewicz et al. 2001]), and good heuristics for selecting the decision literal when applying the Decide rule. As said, conflict analysis procedures for applying Backjump and the possibility of applying learning by other forms of resolution have also been well studied.

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 additional knowledge of the search space compiled into the newly learned lemmas will lead the heuristics for Decide to behave differently, and possibly 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 in theory. Essentially, any Basic DPLL derivation to *FailState* is equivalent to a *tree-like* refutation by resolution. But for some classes of problems tree-like proofs are always exponentially larger than the smallest *general*, that is, DAG-like, resolution ones [Bonet et al. 2000]. The good news is that DPLL with learning and restarts becomes again equivalent to general resolution with respect to such notions of proof complexity [Beame et al. 2003].

In our formalism, restarts can be simply modeled by the following rule:

Definition 2.14. The Restart rule is:

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

Adding the Restart rule to DPLL with Learning, it is obvious that all results of this section hold as long as 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. This is formalized below.

Definition 2.15. Consider a derivation by the DPLL system with learning extended with the Restart rule. We say that Restart *has increasing periodicity* in the derivation if, for each subderivation $S_i \implies \cdots \implies S_j \implies \cdots \implies S_k$ where the steps producing S_i, S_j, and S_k are the only Restart steps, the number of Basic DPLL steps in $S_i \implies \cdots \implies S_j$ is strictly smaller than in $S_j \implies \cdots \implies S_k$.

THEOREM 2.16. *Any derivation $\emptyset \parallel F \implies S_1 \implies \cdots$ by the transition system L extended with the Restart rule is finite if it contains no infinite subderivations consisting of only Learn and Forget steps, and Restart has increasing periodicity in it.*

PROOF. By contradiction, assume Der is an infinite derivation fulfilling the requirements. Let $\succ$ be the well-founded ordering on (the first components of) states defined in the proof of Theorem 2.10. In a subderivation of Der 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 is no infinite subderivation consisting of only Learn and Forget steps, there must be infinitely many Restart steps in Der. Also, if between two states there is at least one Basic DPLL step and no Restart 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 propositional symbols, there cannot be any subderivations with more than n Basic DPLL steps between two Restart steps. This contradicts the fact that there are infinitely many Restart steps if Restart has increasing periodicity in Der. □

In conclusion, in this section, we have formally described a large family of practical implementations of DPLL with learning and restarts, and proved that they provide a decision procedure for propositional satisfiability.

3. *Abstract DPLL Modulo Theories*

For many applications, encoding the problems into propositional logic is not the right choice. Frequently, a better alternative is to express the problems in a richer non-propositional logic, considering satisfiability with respect to a background theory T.

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 [Alur 1999]. 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 [Burch and Dill 1994]. To mention just one further example, the conditions arising from program verification usually involve arrays, lists and other data structures, so it becomes very natural to consider satisfiability problems *modulo* the combined theory T of these data structures. In such applications, typical formulas consist of large sets of clauses such as:

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

containing purely propositional atoms as well as atoms over the combined theory. This is known as the *Satisfiability Modulo Theories* (SMT) problem for a theory T: given a formula F, determine whether F is T-satisfiable, that is, whether there exists a model of T that is also a model of F.

In this section, we show that many of the existing techniques for handling SMT, of which SAT is a particular case if we consider T to be the empty theory, can be described and discussed within the Abstract DPLL framework.

3.1. FORMAL PRELIMINARIES ON SATISFIABILITY MODULO THEORIES. Throughout this section, we consider the same definitions and notation given in Section 2 for the propositional case, 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, here we also consider some notions of first-order logic (see e.g., Hodges [1993]). A *theory* T is a set of closed first-order formulas. 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 in the previous section, a partial assignment M will sometimes also be seen as a conjunction of literals and hence as a formula. If M is a T-consistent partial assignment and F is a formula such that $M \models F$, that is, M is a (propositional) model of F, then we say that *M is a T-model of F*. 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*. A *theory lemma* is a clause C such that $\emptyset \models_T C$.

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, given a background theory T, we will only consider the SMT problem for *ground* (and hence quantifier-free) CNF formulas F. Such formulas may contain *free* constants, that is, constant symbols not in the signature of T, which, as far as satisfiability is concerned, can be equivalently seen as existential variables. Other than free constants, all other predicate and function symbols in the formulas will instead come from the signature of T. From now on, when we say formula we will mean a formula satisfying these restrictions.

We will consider here only theories T such that the T-satisfiability of conjunctions of such ground literals is decidable. We will call any decision procedure for this problem a *T-solver*.

3.2. AN INFORMAL PRESENTATION OF SMT PROCEDURES. The current techniques for deciding the satisfiability of a ground formula F with respect to a background theory T can be broadly divided into two main categories: *eager* and *lazy*.

3.2.1. *Eager SMT Techniques.* In eager techniques, the input formula is translated using a satisfiability-preserving transformation into a propositional CNF formula which is then checked by a SAT solver for satisfiability (see, e.g., Bryant et al. [2001], Bryant and Velev [2002], and Strichman [2002]).

One of the strengths of this eager approach is that it can always use the best available SAT solver off the shelf. When the new generation of efficient SAT solvers such as Chaff [Moskewicz et al. 2001] became available, impressive results using the eager SMT approach were achieved by Bryant's group at CMU with the solver *UCLID* [Lahiri and Seshia 2004] for the verification of pipelined processors.

However, eager techniques are not very flexible: to make them efficient, sophisticated ad-hoc translations are required for each theory. For example, for EUF and for Difference Logic there exist the *per-constraint* encoding [Bryant and Velev 2002; Strichman et al. 2002], the *small domain* encoding (or *range-allocation* techniques), [Pnueli et al. 1999; Bryant et al. 2002; Talupur et al. 2004; Meir and Strichman 2005], and several hybrid approaches [Seshia et al. 2003]. The eager encoding approach can also handle integer linear arithmetic and the theory of arrays (see Seshia [2005]).

In spite of the effort spent in devising efficient translations, on many practical problems the translation process or the SAT solver run out of time or memory (see de Moura and Ruess [2004]). The current alternative techniques explained below are in many cases several orders of magnitude faster.

The correctness of the eager approach for SMT relies on the correctness of both the SAT solver and the translation, which is specific for each theory. It is out of the scope of this article to discuss the correctness of these ad-hoc translations. Assuming them to be correct, the correctness of the eager techniques follows from the results of Section 2.

3.2.2. *Lazy SMT Techniques.* As an alternative to the eager approach, one can use a specialized T-solver for deciding the satisfiability of conjunctions of theory literals. Then, a decision procedure for SMT is easily obtained by converting the given formula into disjunctive normal form (DNF) and using the T-solver to check whether any of the DNF conjuncts is satisfiable. However, the exponential blowup usually caused by the conversion into DNF makes this approach too inefficient.

A lot of research has then looked into ways to combine the strengths of specialized T-solvers with the strengths of state-of-the-art SAT solvers in dealing with the Boolean structure of formulas. The most widely used approach in the last few years is usually referred to as the *lazy* approach [Armando et al. 2000; Filliâtre et al. 2001; Audemard et al. 2002; Barrett et al. 2002; de Moura and Rueß 2002; Flanagan et al. 2003; Armando et al. 2004; Ball et al. 2004]. In this approach, each atom occurring in a formula F to be checked for satisfiability is initially considered simply as a propositional symbol, *forgetting* about the theory T. Then the formula is given to a SAT solver. If the SAT solver determines it to be (propositionally) unsatisfiable, then F is T-unsatisfiable as well. If the SAT solver returns instead a propositional model M of F, then this assignment (seen as a conjunction of literals) is checked by a T-solver. If M is found T-consistent then it is a T-model of F. Otherwise, the T-solver builds a ground clause that is a logical consequence of T, that is, a theory lemma, precluding that assignment. This lemma is added to F and the SAT solver is started again. This process is repeated until the SAT solver finds a T-model or returns unsatisfiable.

Example 3.1. Assume we are deciding with a lazy procedure the T-satisfiability of a large EUF formula, where T is the theory of equality, and assume that the model M found by the SAT solver contains, among many others, the four literals:

$$b = c, \quad f(b) = c, \quad a \neq g(b), \quad g(f(c)) = a.$$

Then the T-solver detects that M is not a T-model, since

$$b = c \;\wedge\; f(b) = c \;\wedge\; g(f(c)) = a \quad \models_T \quad a = g(b).$$

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 is usually not a good idea as the generated clause may end up containing thousands of literals. Lazy procedures are much more efficient if the T-solver is able instead to generate a small *explanation* of the T-inconsistency of M. In this example, the explanation could be simply the clause $b \neq c \;\vee\; f(b) \neq c \;\vee\; g(f(c)) \neq a \;\vee\; a = g(b)$.

The main advantage of the lazy approach is its flexibility, since it can easily combine any SAT solver with any T-solver. More importantly, if the SAT solver used by the lazy SMT procedure is based on DPLL, then several refinements exist that make the SMT procedure much more efficient. Here we outline the most significant ones.

Incremental T-solver. The T-consistency of the assignment can be checked incrementally, while the assignment is being built by the DPLL procedure, without delaying the check until a propositional model has been found. This can save a large amount of useless work. It can be done fully eagerly, detecting T-inconsistencies as soon as they are generated, or, if that is too expensive, at regular intervals, for example, once every k literals added to the assignment. The idea was already mentioned in Audemard et al. [2002] under the name of *early pruning* and in Barrett [2003] under the name of *eager notification*. Currently, most SMT implementations work with incremental T-solvers. The incremental use of T-solvers poses different requirements on their implementation: to make the incremental approach effective in practice, the solver should (on average, say) be faster in processing one additional input literal l than in re-processing from scratch all previous inputs and l together. For many theories this can indeed be done; see, for example, Section 4.3, where we describe an incremental solver for Difference Logic.

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 some point where the assignment was still T-consistent, instead of restarting the search from scratch. For instance, if, in Abstract DPLL terms, the current state is of the form $M \parallel F$ and M has been detected to be T-inconsistent, then there is some subset $\{l_1 \cdots l_n\}$ of M such that $\neg l_1 \vee \cdots \vee \neg l_n$ is a theory lemma. This lemma can be added to the clause set, and, since it is conflicting, that is, it is false in M, Backjump or Fail can be applied. As we will formally prove below, after the backjump step this lemma is no longer needed for completeness and could be safely forgotten: the procedure will search through all propositional models, finding a T-consistent one whenever it exists. Nevertheless, keeping theory lemmas can still be very useful for efficiency reasons, because it may cause an important amount of pruning later in the search. Theory lemmas are especially effective if they are small, as observed in, for example, de Moura and Rueß [2002] and Barrett [2003]. On-line SAT solvers (in combination with incremental T-solvers) are now common in SMT implementations, and state-of-the-art SAT solvers like zChaff or MiniSAT provide this functionality.

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. To overcome this limitation, the T-solver can also be used in a given DPLL state $M \parallel F$ to detect literals l occurring in F such that $M \models_T l$, allowing the DPLL procedure to move to the state $M\, l \parallel F$. We call this process *theory propagation*.

The idea of theory propagation was first mentioned in Armando et al. [2000] under the name of *Forward Checking Simplification*, and since then it has been applied, in limited form, in very few other systems (see Section 5). In contrast, theory propagation plays a major role in the DPLL(T) approach, introduced in Section 4 of this article. There we show that, somewhat against expectations, practical T-solvers can be designed to include this feature in an efficient way. A highly non-trivial issue is how to perform conflict analysis appropriately in the context of theory propagation. Different options and possible problems for doing this are analyzed and solved in detail in Section 5, something that, to our knowledge, had not been done before. In Section 6, we show that theory propagation, if handled well, has a crucial impact on the performance of SMT systems.

Exhaustive Theory Propagation. For some theories, it even pays off to perform *all* possible Theory Propagations before applying the Decide rule. This idea of exhaustive theory propagation is also introduced in the DPLL(T) approach presented here.

Lazy techniques that learn theory lemmas and do not perform any theory propagation in effect dump a large number of ground consequences of the theory into the clause set, duplicating theory information into the SAT solver. This duplication is instead completely unnecessary in a system with exhaustive theory propagation—and is greatly reduced with non-exhaustive theory propagation.

The reason is that any literal generated by unit propagation over a theory lemma can also be generated by theory propagation.[1]

For some logics, such as Difference Logic, for instance, exhaustive theory propagation usually yields speedups of several orders of magnitude, as we show in Section 6.

Using an incremental T-solver in combination with an on-line SAT solver is known to be crucial for efficiency. Possibly with the only exception of Verifun [Flanagan et al. 2003], an experimental system no longer under development, most, if not all, state-of-the-art SMT systems use incremental solvers. On the other hand, only a few SMT systems so far use theory propagation, as we will discuss in Section 5.

[1] But see the discussion about strategies with lazier theory propagation at the end of Section 5.1.

3.3. ABSTRACT DPLL MODULO THEORIES. In this section, we formalize the different enhancements of the lazy approach to Satisfiability Modulo Theories. We do this by adapting the Abstract DPLL framework for the propositional case presented in the previous section. One significant difference is that here we deal with ground first-order literals instead of propositional ones. Except for that, the rules Decide, Fail, UnitPropagate, and Restart remain unchanged: they will still regard all literals as syntactical items as in the propositional case. Only Learn, Forget and Backjump are slightly modified to work modulo theories: in these rules, entailment between formulas now becomes entailment in T. In addition, atoms of T-learned clauses can now also belong to M, and not only to F; this is required for Property 3.9 below, needed to recover from T-inconsistent states. Note that the theory version of Backjump below uses both the propositional notion of satisfiability ($\models$) and the first-order notion of entailment modulo theory ($\models_T$).

Definition 3.2. The rules T-Learn, T-Forget and T-Backjump are:

T-Learn:

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

T-Forget:

$$M \parallel F,\ C \qquad \Longrightarrow \quad M \parallel F \qquad \textbf{if} \begin{cases} F \models_T C \end{cases}$$

T-Backjump:

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

3.3.1. *Modeling the Naive Lazy Approach.* Using these rules, it is easy to model the basic lazy approach (without any of the refinements of incremental T-solvers, on-line SAT solvers or theory propagation). Each time a state $M \parallel F$ is reached that is final with respect to Decide, Fail, UnitPropagate, and T-Backjump, that is, 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, as we will prove below. If M is not T-consistent, then there exists a subset $\{l_1, \ldots, l_n\}$ of M such that $\emptyset \models_T \neg l_1 \vee \cdots \vee \neg l_n$. By one T-Learn step, the theory lemma $\neg l_1 \vee \cdots \vee \neg l_n$ can be learned and then Restart can be applied. As we will prove below, if these learned theory lemmas are never removed by the T-Forget rule, this strategy is terminating under similar requirements as those in the previous section, namely, the absence of infinite subderivations consisting of only Learn and Forget steps and the increasing periodicity of Restart steps. Then, the strategy is also sound and complete as stated in the previous section: the initial formula is T-unsatisfiable if, and only if, *FailState* is reached; moreover, if *FailState* is not reached then a T-model has been found.

3.3.2. *Modeling the Lazy Approach with an Incremental T-Solver.* Assume a state $M \parallel F$ has been reached where M is T-inconsistent. Note that in practice this is detected by the incremental T-solver, and that this state need not be final now. Then, as in the naive lazy approach, there exists a subset $\{l_1, \ldots, l_n\}$ of M such that $\emptyset \models_T \neg l_1 \vee \cdots \vee \neg l_n$. This theory lemma is then learned, producing the state $M \parallel F,\ \neg l_1 \vee \cdots \vee \neg l_n$. As in the previous case, Restart can then be applied and the same results hold.

3.3.3. *Modeling the Lazy Approach with an Incremental T-Solver and an on-Line SAT Solver.* As in the previous case, if a subset $\{l_1, \ldots, l_n\}$ of M is detected such that $\emptyset \models_T \neg l_1 \vee \cdots \vee \neg l_n$, the theory lemma is learned, reaching the state $M \parallel F,\ \neg l_1 \vee \cdots \vee \neg l_n$. But now, since in addition we consider an online SAT solver, instead of completely restarting, the procedure *repairs* the T-inconsistency of the partial assignment by exploiting the fact that the recently learned theory lemma is a conflicting clause. As we show later, and similarly to what happened in the propositional case, if there is no decision literal in M then Fail applies, otherwise T-Backjump applies. Our results below prove that, even if the theory lemma is always forgotten immediately after backjumping, this approach is terminating, sound, and complete under similar conditions as the ones of the previous section.

3.3.4. *Modeling the Previous Refinements and Theory Propagation.* This requires the following additional rule:

Definition 3.3. The TheoryPropagate rule is:

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

The purpose of this rule is to prune the search by assigning a truth value to literals that are (propositionally) undefined by the current assignment M but T-entailed by it, rather than letting the Decide rule guess a value for them. As said, this sort of propagation can lead to dramatic improvements in performance. Below we prove that the correctness results mentioned for the previous three lazy approaches also hold in combination with arbitrary applications of this rule.

3.3.5. *Modeling the Previous Refinements and Exhaustive Theory Propagation.* Exhaustive theory propagation is modeled simply by assuming that TheoryPropagate is applied with a higher priority than Decide. The correctness of this approach follows immediately from the correctness of the previous one which had arbitrary applications of TheoryPropagate.

3.4. Correctness of Abstract DPLL Modulo Theories.

Up to now, we have seen several different application strategies of (subsets) of the given rules, which lead to different SMT procedures. In this subsection we give a simple and uniform proof showing that all the approaches described in the previous subsection are indeed decision procedures for the SMT problem. The proofs are structured in the same way as the ones given in Section 2.5 for the propositional case, and hence here we focus on the variations and extensions that are needed.

Definition 3.4. The *Basic DPLL Modulo Theories system* consists of the rules Decide, Fail, UnitPropagate, TheoryPropagate, and T-Backjump.

Definition 3.5. The *Full DPLL Modulo Theories system*, denoted by FT, consists of the rules of Basic DPLL Modulo Theories and the rules T-Learn, T-Forget, and Restart.

As before, a decision procedure will be obtained by generating a derivation using the given rules with a particular strategy. However, here the aim of a derivation is to compute a state S to which the main theorem of this section, Theorem 3.10, can be applied, that is, a state S such that: (i) S is final with respect to the rules of Basic DPLL Modulo Theories and (ii) if S is of the form $M \parallel F$ then M is T-consistent.

Property 3.9 below provides a very general class of strategies in which such a state S is always reached, without violating the requirements of termination of Theorem 3.7 (also given below). Such a state S can be recognized in a similar way as in the propositional case: it is either *FailState* or it is of the form $M \parallel F$ where all the literals of F are defined in M, there are no conflicting clauses, and M is T-consistent.

The following lemma states invariants similar to the ones of Lemma 2.7 of the previous section.

LEMMA 3.6. *If* $\emptyset \parallel F \Longrightarrow^{*}_{\mathrm{FT}} M \parallel G$, *then the following hold:*

(1) *All the atoms in* M *and all the atoms in* G *are atoms of* F.

(2) M *contains no literal more than once and is indeed an assignment, that is, it contains no pair of literals of the form* p *and* $\neg p$.

(3) G *is* T*-equivalent to* F.

(4) *If* M *is of the form* $M_0\ l_1\ M_1\ \cdots\ l_n\ M_n$, *where* $l_1, \ldots, l_n$ *are all the decision literals of* M, *then* $F, l_1, \ldots, l_i \models_T M_i$ *for all* i *in* $0 \cdots n$.

PROOF. As for Lemma 2.7, all rules preserve the properties. The new rule TheoryPropagate preserves them like UnitPropagate; the other rules as for their propositional versions. □

THEOREM 3.7 (TERMINATION). *Let* Der *be a derivation of the form:*
$\emptyset \parallel F = S_0 \Longrightarrow_{\mathrm{FT}} S_1 \Longrightarrow_{\mathrm{FT}} \cdots$
Then Der *is finite if the following two conditions hold:*

(1) Der *has no infinite subderivations consisting of only* T*-Learn and* T*-Forget steps.*

(2) *For every subderivation of* Der *of the form:*
$S_{i-1} \Longrightarrow_{\mathrm{FT}} S_i \Longrightarrow_{\mathrm{FT}} \cdots \Longrightarrow_{\mathrm{FT}} S_j \Longrightarrow_{\mathrm{FT}} \cdots \Longrightarrow_{\mathrm{FT}} S_k$
where the only three Restart *steps are the ones producing* S_i, S_j, *and* S_k, *either:*
—*there are more Basic DPLL Modulo Theories steps in* $S_j \Longrightarrow_{\mathrm{FT}} \cdots \Longrightarrow_{\mathrm{FT}} S_k$ *than in* $S_i \Longrightarrow_{\mathrm{FT}} \cdots \Longrightarrow_{\mathrm{FT}} S_j$, *or*
—*a clause is learned*[2] *in* $S_j \Longrightarrow_{\mathrm{FT}} \cdots \Longrightarrow_{\mathrm{FT}} S_k$ *that is not forgotten in* Der.

PROOF. The proof is a slight extension of the one of Theorem 2.16. The only new aspect is that some Restart steps are applied with non-increasing periodicity. But since for each one of them a new clause has been learned that is never forgotten in Der, there can only be finitely many of them. From this, a contradiction follows as in Theorem 2.16. □

[2] See Definition 2.5.

LEMMA 3.8. *If* $\emptyset \parallel F \Longrightarrow^*_{\mathrm{FT}} M \parallel F'$ *and there is some conflicting clause in* $M \parallel F'$*, that is,* $M \models \neg C$ *for some clause* C *in* F'*, then either* Fail *or* T-Backjump *applies to* $M \parallel F'$.

PROOF. As in Lemma 2.8. □

PROPERTY 3.9. *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.*

PROOF. If M is T-inconsistent, then there exists a subset $\{l_1, \ldots, l_n\}$ of M such that $\emptyset \models_T \neg l_1 \vee \cdots \vee \neg l_n$. Hence, the conflicting clause $\neg l_1 \vee \cdots \vee \neg l_n$ is either in $M \parallel F'$, or else it can be learned by one T-Learn step. □

Lemma 3.8 and Property 3.9 show that a rule of Basic DPLL modulo theories is always applicable to a state of the form $M \parallel F$, or to its successor after a single T-Learn step, whenever a literal of F is undefined in M, or F contains a conflicting clause, or M is T-inconsistent. Together with Theorem 3.7 (Termination), this shows how to compute a state to which the following main theorem is applicable.

THEOREM 3.10. *Let* Der *be a derivation* $\emptyset \parallel F \Longrightarrow^*_{\mathrm{FT}} S$ *, where (i)* S *is final with respect to Basic DPLL Modulo Theories, and (ii) if* S *is of the form* $M \parallel F'$ *then* M *is* T*-consistent. Then*

(1) *S is FailState if, and only if,* F *is* T*-unsatisfiable.*

(2) *If S is of the form* $M \parallel F'$*, then* M *is a* T*-model of* F*.*

PROOF. The first result follows from Lemmas 3.6(3), 3.6(4), as in Theorem 2.12. The second part is proved as in Lemma 2.9 of the previous section, but using Lemma 3.8 and Lemma 3.6(3), instead of Lemma 2.8 and Lemma 2.7(3). □

The previous theorem shows that a large family of practical approaches provide a decision procedure for satisfiability modulo theories. Note that the results of this section are independent from the theory T under consideration, the only (obviously necessary) requirement being the decidability of the T-consistency of conjunctions of ground literals.

We conclude this section by observing that, as in the propositional case, our definition of final state for Abstract DPLL Modulo Theories forces the assignment M in a state of the form $M \parallel G$ to be total. We remarked in the previous section that the alternative definition of final state where M can be partial as long as it satisfies G is inefficient in practice in the SAT case. With theories, however, this is not always true. Depending on the theory T and the available T-solver, it may be considerably more expensive to insist on extending a satisfying partial assignment to a total one than to check periodically whether the current assignment has become a model of the current formula. The reason is that by Theorem 3.10 one can stop the search with a final state $M \parallel G$ only if M is also T-consistent, and T-consistency checks can have a high cost, especially when the T-satisfiability of conjunction of literals is NP-hard. We have maintained the same definition of final state for both Abstract DPLL and Abstract DPLL Modulo Theories mainly for simplicity, to make the lifting of the former to the latter clearer. We stress though that as in the previous section essentially the same correctness proof applies if one uses the alternative definition of final state in this section.

4. *The DPLL(T) Approach*

We have now seen an abstract framework that allows one to model a large number of complete and terminating strategies for SMT. In this section, we describe the DPLL(T) approach for Satisfiability Modulo Theories, a general modular architecture on top of which actual implementations of such SMT strategies can be built. This architecture 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 DPLL(T) system for T is produced by instantiating the parameter X with a module Solver_T that can handle conjunctions of literals in T, that is, a T-solver.

The basic idea is similar to the *CLP*(X) scheme for constraint logic programming [Jaffar and Maher 1994]: 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 DPLL(T) architecture presented here combines the advantages of the eager and lazy approaches to SMT. On the one hand, the architecture allows for very efficient implementations, as witnessed by our system, Barcelogic, which implements DPLL(T) for a number of theories and compares very favorably with other SMT systems—see Section 6. 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 the general DPLL(X) engine, provided that these solvers conform to a minimal interface.

4.1. OVERALL ARCHITECTURE OF DPLL(T). At each state $M \parallel F$ of a derivation, the DPLL(X) engine knows M and F, but it treats all literals and clauses as purely propositional ones. As a consequence, all the needed theory-based inferences are exclusively localized in the theory solver Solver_T, which knows M but not the current F.

For the purposes of this article, it is not necessary to precisely define the interface between DPLL(X) and Solver_T. It suffices to know that Solver_T provides operations that can be called by DPLL(X) to:

—Notify Solver_T that a certain literal has been set to true.

—Ask Solver_T to check whether the current partial assignment M, as a conjunction of literals, is T-inconsistent. This request can be made by DPLL(X) with different degrees of *strength*: for theories where deciding T-inconsistency is in general expensive, it might be more convenient to use cheaper, albeit incomplete, T-inconsistency checks for most of the derivation, and resort to a more expensive but complete check only when necessary.[3]

It is required that when Solver_T detects a T-inconsistency it is also able to identify a subset $\{l_1, \ldots, l_n\}$ of M such that $\emptyset \models_T \neg l_1 \vee \cdots \vee \neg l_n$. This theory lemma $\neg l_1 \vee \cdots \vee \neg l_n$, which we will call the *(theory) explanation* of the T-inconsistency, is then communicated by Solver_T to the engine.

[3] Note that, according to the correctness results of Abstract DPLL modulo theories, a *decision* of T-inconsistency is only needed when a final state with respect to the Basic DPLL rules is reached.

—Ask *Solver*$_T$ to identify currently undefined input literals that are T-consequences of M. Again, this request can be made by DPLL(X) with different degrees of strength. *Solver*$_T$ answers with a (possibly empty) list of literals of the input formula that are newly detected T-consequences of M. Note that for this operation *Solver*$_T$ needs to know the set of input literals.

—Ask *Solver*$_T$ to provide a justification for the T-entailment of some theory propagated literal l. This is needed for the following reasons. In a concrete implementation of the DPLL(X) engine, backjumping is typically guided by a conflict graph, as explained in Example 2.6. But there is a difference with respect to the purely propositional case: a literal l at a node in the graph can now also be due to an application of theory propagation. Hence, building the graph requires that *Solver*$_T$ be able to recover and return as a justification of l a (preferably small, non-redundant) subset $\{l_1, \ldots, l_n\}$ of literals of the assignment M that T-entailed l when l was T-propagated. Computing that subset amounts to generating the theory lemma $\neg l_1 \vee \cdots \vee \neg l_n \vee l$. We will call this lemma the *(theory) explanation* of l. (See Example 5.1, and also Section 4.3 and Section 5 for more details and refinements.)

—Ask *Solver*$_T$ to undo the last n notifications that a literal has been set to true.

In the rest of this section, we describe two concrete SMT strategies for the Abstract DPLL modulo theories framework, and show how they can be implemented using the DPLL(T) architecture.

The first one, described in Section 4.2, performs exhaustive theory propagation in a very eager way: in a state $M \parallel F$, TheoryPropagate is immediately applied whenever some input literal l is T-entailed by M. Therefore, *Solver*$_T$ is required to detect *all* such entailments immediately after a literal is set to true. In contrast, the second DPLL(T) system, described in Section 4.4, allows *Solver*$_T$ to sometimes fail to detect some entailed literals.

Each system is accompanied by a concrete motivating example of a theory of practical relevance, namely Difference Logic and EUF Logic, respectively. For Difference Logic, an efficient design for *Solver*$_T$ is described in Section 4.3.

Further refinements of theory propagation and conflict-driven clause learning are discussed in more detail in Section 5.

4.2. DPLL(T) WITH EXHAUSTIVE THEORY PROPAGATION AND DIFFERENCE LOGIC. Here, we deal with a particular application strategy of the rules of Abstract DPLL Modulo Theories modeling exhaustive theory propagation. We show how it can be implemented using the DPLL(T) architecture, and explain the roles of the DPLL(X) engine and the theory solver *Solver*$_T$ in it.

Solver$_T$ processes the input formula, stores the list of all literals occurring in it, and hands it over to DPLL(X), which treats it as a purely propositional CNF. After that, the various Abstract DPLL rules are applied by DPLL(X) as described below:

TheoryPropagate. Immediately after *Solver*$_T$ is notified that a literal l has been added to M, (e.g., as a consequence of UnitPropagate or Decide), *Solver*$_T$ is also requested to provide *all* input literals that are T-consequences of $M\,l$ but not of M alone. Then, for each one of them, TheoryPropagate is immediately applied by DPLL(X). Note that, this way, M never becomes T-inconsistent, a property that can be exploited by *Solver*$_T$ to increase its efficiency (see the next subsection for the case of Difference Logic), and by the DPLL(X) engine since it will never need to ask for the T-consistency of M.

UnitPropagate. If TheoryPropagate is not applicable, DPLL(X) tries to apply UnitPropagate next, possibly triggering more rounds of theory propagation, and stops if it discovers a conflicting clause. (In a concrete implementation all this can be implemented with the commonly used two-watched-literals scheme.)

Backjump and Fail. If DPLL(X) detects a conflicting clause, it immediately applies T-Backjump or Fail, depending respectively on whether the current assignment contains a decision literal or not. (In a concrete implementation, an appropriate backjump clause can be computed as explained in the next section.) At each backjump, DPLL(X) tells $Solver_T$ how many literals have been removed from the assignment.

T-Learn. Immediately after each T-Backjump application, the T-Learn rule is applied to learn the backjump clause. This is possible because this clause is always a T-consequence of the formula F in the current state $M \parallel F$. Note that, as explained in Section 3.2 for the case of exhaustive theory propagation, theory lemmas (clauses C such that $\emptyset \models_T C$) are never learned, since they are useless in this context.

Restart. For correctness with respect to the Abstract DPLL modulo theories framework, one must guarantee that Restart has increasing periodicity. Typically, this is achieved by only applying Restart when certain system parameters reach some prescribed limits, such as the number of conflicts or the number of new units derived, and increasing this restart limit periodically.

T-Forget. For correctness with respect to Abstract DPLL modulo theories, it suffices to apply this rule only to previously T-learned clauses. This is what is usually done, removing part of these clauses according to their activity (e.g., the number of times involved in recent conflicts).

Decide. In this strategy, DPLL(X) applies Decide only if none of the other Basic DPLL rules apply. The choice of the decision literal is well known to have a strong impact on the search behavior. Numerous heuristics for this purpose exist.

4.3. DESIGN OF $Solver_T$ FOR DIFFERENCE LOGIC.

To provide an example in this article of $Solver_T$ for a given T, here we briefly outline the design of a theory solver for Difference Logic. 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.

In Difference Logic, the background theory T can be the theory of the integers, the rationals or the reals, depending on the application. Input formulas are restricted to Boolean combinations of atoms of the form $a \leq b + k$, where a and b are free constants and k is a (possibly negative) integer, rational or real constant. Over the integers, atoms of the form $a < b + k$ can be equivalently written as $a \leq b + (k - 1)$; for instance, $a < b + 7$ becomes $a \leq b + 6$. A similar transformation exists for the rationals and reals, by decreasing k by a small enough amount ε. For a given input formula, the ε to be applied to its literals can be computed in linear time [Schrijver 1987; Armando et al. 2004]. Similarly, negations and equalities can also be removed, and one can assume that all literals are of the form $a \leq b + k$. Their conjunction can be seen as a weighted graph G with an edge $a \xrightarrow{k} b$ for each literal $a \leq b + k$. Independently of whether T is the theory of the integers, the rationals or the reals, such a conjunction is T-satisfiable if, and only if, there is no cycle in G with negative accumulated weight. Therefore, once all literals are of the form $a \leq b + k$, the specific theory does not matter any more.

4.3.1. *Initial Setup.* As said, 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) 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, over the integers, if $a \leq b + 2$ and $b \leq a - 3$ occur in the input, then, since one is equivalent to 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 an ordering on the free constants and require that the smallest one, say a in the example above, be always on the left-hand side of the $\leq$ symbol. So here we would have that $b \leq a - 3$ is handled as the negation of $a \leq b + 2$.

For reasons we will see below, *Solver*$_T$ also builds a data structure containing, for each constant symbol, the number of input literals it occurs in, and the list of all these literals.

4.3.2. *DPLL(X) Sets the Truth Value of a Literal.* Then, *Solver*$_T$ adds the corresponding edge to the graph. Here we will write $a_0 \xrightarrow{k\,*} a_n$ if there is a path in the graph of the form $a_0 \xrightarrow{k_1} a_1 \xrightarrow{k_2} \cdots \xrightarrow{k_{n-1}} a_{n-1} \xrightarrow{k_n} a_n$ with $n \geq 0$ and where $k = k_1 + \cdots + k_n$ is called the length of this path.

Note that one can assume that DPLL(X) does not communicate to *Solver*$_T$ any redundant edges (i.e., edges already entailed by G), 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 $a_i \leq b_j + (k_i + k + k'_j)$, that is, 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, the graph is kept in double adjacency list representation. 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). Its cost, for M literals containing N different constant symbols, is $O(N \cdot M)$.

While doing this, the visited nodes are marked and inserted into two lists, one for the a_i's and one for the b_j's. At the same time, two counters are kept measuring the total number of input literals containing the a_i's and, respectively, the b_j's.

Then, if, without loss of generality, the a_i's are the ones that occur in less input literals, we check, for each input literal l containing some a_i, whether the other constant in l is some of the found b_j, and whether l is entailed or not (this can be checked in constant time since previously all b_j have been marked). The asymptotic worst-case cost of this part is $O(L)$, where L is the number of different input literals. In our experience, this is much faster than the $O(N^2)$ check of the Cartesian product of a_i's and b_j's.

4.3.3. *Implementation of Explain and Backtrack.* Whenever the mth edge is added to the graph, the edge is annotated with its *insertion number m*. When a literal l of the form $a \leq b + k$ is returned as a T-consequence of the mth edge, this m is recorded together with l. If later on the explanation for l is required, a path in the graph from a to b of length at most k is searched, using a depth-first search as before, but without traversing any edges with insertion number greater than m. This not only improves efficiency, but it is also needed for not returning "too new" explanations, which may create cycles in the implication graph (see Section 5). Each time DPLL(X) backjumps, it communicates to $Solver_T$ how many edges it has to remove, for example, up to some insertion number m. According to our experiments, the best way (with negligible cost) for dealing with this in $Solver_T$ is the naive one, that is, using a trail stack of edges with their insertion numbers and all their associated T-consequences.

4.4. DPLL(T) WITH NONEXHAUSTIVE THEORY PROPAGATION AND EUF LOGIC. For some logics, such as the logic of Equality with Uninterpreted Functions (EUF), exhaustive theory propagation is not the best strategy. In EUF, atoms consist of ground equations between terms, and the theory T consists of the axioms of reflexivity, symmetry, and transitivity of '=', as well as the *monotonicity* axioms, saying, for all f, that $f(x_1 \cdots x_n) = f(y_1 \cdots y_n)$ whenever $x_i = y_i$ for all i in $1 \cdots n$ (see also Example 3.1).

Our experiments with EUF revealed that a nonexhaustive strategy behaves better in practice than one with exhaustive theory propagation. More precisely, we found that detecting exhaustively all *negative* equality consequences is very expensive, whereas all positive equalities can propagated efficiently by means of a congruence closure algorithm [Downey et al. 1980]. It is beyond the scope of this article to describe the design of a theory solver for EUF. We refer the reader to Nieuwenhuis and Oliveras [2003] for a description and discussion of a modern incremental, backtrackable congruence closure algorithm for this purpose. We point out that efficiently retrieving explanations (for constructing the conflict graph and generating theory lemmas) inside an incremental congruence closure algorithm is nontrivial. Increasingly better techniques have been developed in de Moura et al. [2004] Stump and Tan [2005], and Nieuwenhuis and Oliveras [2005b].

We describe below an application strategy of Abstract DPLL Modulo Theories for DPLL(T) with nonexhaustive theory propagation. The emphasis will be on those aspects that differ from the exhaustive case, and on how and when T-inconsistent partial assignments M are detected and repaired.

TheoryPropagate. In this strategy, when $Solver_T$ is asked for new T-consequences, it may return only an incomplete list. Therefore, DPLL(X) can no longer maintain the invariant that the partial assignment is always T-consistent as in the exhaustive case of Section 4.2. For this reason, it is no longer necessary to ask for T-consequences as eagerly as in the exhaustive case. Instead, for efficiency reasons it is better to ask $Solver_T$ for new T-consequences only if no Basic DPLL rule other than Decide applies and the current assignment is T-consistent. For each returned T-consequence, TheoryPropagate is immediately applied by DPLL(X).

UnitPropagate. DPLL(X) applies this rule while possible unless it detects a conflicting clause.

Backjump and Fail. DPLL(X) may apply T-Backjump or Fail due to two possible situations. The first one is when it detects a conflicting clause, as usual. The second one is due to a T-inconsistency of the current partial assignment M.

Solver$_T$ is asked to check the T-consistency of M each time no Basic DPLL rule other than Decide applies—and before being asked for theory consequences. When M is T-inconsistent *Solver*$_T$ identifies a subset $\{l_1, \ldots, l_n\}$ of it such that $\emptyset \models_T \neg l_1 \vee \cdots \vee \neg l_n$, and returns the theory lemma $\neg l_1 \vee \cdots \vee \neg l_n$ as an explanation of the inconsistency. *DPLL*(X) then handles the lemma as a conflicting clause, applying T-Backjump or Fail to it.

***T*-Learn.** Immediately after each T-Backjump application, the T-Learn rule is applied for learning the backjump clause.

Now, in backjumps due to T-inconsistencies, the backjump clause may sometimes be the theory lemma denoting the T-inconsistency itself (if it has only one literal of the current decision level). Therefore, in this case, sometimes theory lemmas will be learned. Another possibility is to always learn the theory lemma coming from a T-inconsistency, even if it is not the backjump clause. This may be useful, because it prevents the same T-inconsistency from occurring again.

Restart. This rule is applied as in the exhaustive strategy of Section 4.2.

***T*-Forget.** It 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. This is again less simple than in the exhaustive case, because different forgetting policies could be applied to the two kinds of lemmas. Note that, in any case, none of the lemmas needs to be kept for completeness.

Decide. This rule is applied as in the exhaustive strategy of Section 4.2.

We conclude this section by summarizing the key differences between the two strategies for exhaustive and nonexhaustive theory propagation, described in Sections 4.2 and 4.4: in the former, which we applied to Difference Logic, the partial model never becomes T-inconsistent, since *all* input literals that are T-consequences are immediately set to true. In contrast, the DPLL(T) system described in Section 4.4, and applied to EUF logic, allows, for efficiency reasons, *Solver*$_T$ to fail sometimes to detect some entailed literals, and hence it must be able to recover from T-inconsistent partial assignments.

5. *Theory Propagation Strategies and Conflict Analysis*

The idea of theory propagation was first mentioned in Armando et al. [2000] under the name of *Forward Checking Simplification*, in the context of temporal reasoning. The authors suggest that a literal l can be propagated if $\neg l$ is inconsistent with the current state, but they also imply that this is expensive "since it requires a number of T-consistency checks roughly equal to the number of literals in [the whole formula] φ". A similar notion called *Enhanced Early Pruning* is mentioned in Audemard et al. [2002] in the context of the MathSAT system, but nothing is said about when and how it is applied, and how it relates to conflict analysis. Also, the new system Yices (see Section 6) appears to apply some form of theory propagation. Except for these systems and ours, and a forthcoming version of CVC Lite based on the work described here, we are not aware of any other systems that apply theory propagation, nor of any other descriptions of theory propagation in the literature outside our own previous work on the subject.

We remark that the techniques proposed in, for example, Armando et al. [2004] and Flanagan et al. [2003], where the input formula is statically augmented with theory lemmas encoding transitivity constraints, may have effects similar to theory propagation. However, eagerly learning all such constraints is usually too expensive and typically only a subset of them is ever used at run-time.

In Ganzinger et al. [2004], we showed that, somewhat against expectations, practical T-solvers can be designed to do theory propagation efficiently. To the best of our knowledge, before that, the methods for detecting a theory consequence l were essentially based on sending $\neg l$ to the theory solver, and checking whether a T-inconsistency was derived.

Some essential and nontrivial issues about theory propagation have remained largely unstudied until now:

—when to compute the explanations for the theory propagated literals;

—how to handle conflict analysis adequately in the context of theory propagation;

—how eagerly to perform theory propagation.

In this section, we analyze these issues in detail. We point out that thinking in terms of Abstract DPLL Modulo Theories was crucial in giving us a sufficient understanding for doing this analysis in the first place, especially by helping us clearly separate correctness concerns from efficiency ones. We start with a running example illustrating some of the questions above.

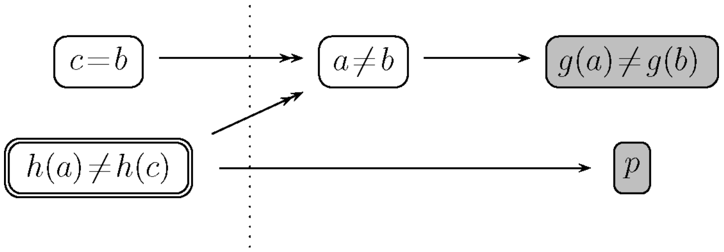
Example 5.1. Consider EUF logic and a clause set F containing, among others:

$$\begin{array}{l}(1)\ a = b \ \vee\ g(a) \neq g(b)\\ (2)\ h(a) = h(c) \ \vee\ p\\ (3)\ g(a) = g(b) \ \vee\ \neg p.\end{array}$$

Now consider a state of the form $M,\ c = b,\ f(a) \neq f(b) \parallel F$, and the following sequence of derivation steps:

Step:	**New literal:**	**Reason:**
Decide	$h(a) \neq h(c)$	
TheoryPropagate	$a \neq b$	since $h(a) \neq h(c) \wedge c = b \models_T a \neq b$
UnitPropagate	$g(a) \neq g(b)$	because of $a \neq b$ and Clause 1
UnitPropagate	p	because of $h(a) \neq h(c)$ and Clause 2.

In the resulting state, Clause 3 is conflicting. When seen as a conflict graph, as done in Example 2.6 for the propositional case, the situation looks as follows:



In this graph, the double arrows $\twoheadrightarrow$ indicate theory propagations, whereas the single arrows denote unit propagations. The backjump clause $h(a) = h(c) \;\vee\; c \neq b$ can be produced by considering the indicated cut in the graph, as in Example 2.6. This clause can also be obtained by the backwards resolution process on the conflicting clause illustrated in Example 2.6, specifically, by resolving in reverse chronological order with the clauses that caused propagations, until a clause with exactly one literal from the current decision level is derived.

The only difference here with respect to the propositional case is that now we can have theory propagated literals as well. For each one of these literals, resolution is done with the theory lemma explaining its propagation (here, the leftmost premise of the last resolution step):

$$\dfrac{h(a)=h(c)\vee c\neq b\vee \mathbf{a}\neq\mathbf{b} \qquad \dfrac{a=b\vee \mathbf{g(a)}\neq\mathbf{g(b)} \qquad \dfrac{h(a)=h(c)\vee \mathbf{p} \qquad g(a)=g(b)\vee\neg\mathbf{p}}{\mathbf{g(a)}=\mathbf{g(b)}\vee h(a)=h(c)}}{h(a)=h(c)\vee \mathbf{a}=\mathbf{b}}}{h(a)=h(c)\;\vee\; c\neq b}$$

The resulting clause can be used as a backjump clause, in the same way as in Example 2.6 for the propositional case.

In what follows, we argue that in general it is not a good idea to compute these theory lemmas (or *explanations*) immediately, during theory propagation. Instead, it is usually better to compute each of them only as needed in resolution steps during conflict analysis. We also explain what problems may occur in delaying the computation of explanations until they are really needed, and give detailed results showing when and how a backjump clause can be found. □

5.1. WHEN TO COMPUTE EXPLANATIONS FOR THE THEORY PROPAGATED LITERALS. Each time a theory propagation step of the form $M \parallel F \implies M\, l \parallel F$ takes place, this is because $l_1 \wedge \cdots \wedge l_n \models_T l$ for some subset $\{l_1, \ldots, l_n\}$ of M. Now, a very simple way of managing theory propagated literals for the purposes of conflict analysis is to use T-Learn immediately after each such a theory propagation step, adding the corresponding theory lemma $\neg l_1 \vee \cdots \vee \neg l_n \vee l$ to the current formula. After that, the theory propagated literal l can be simply seen as a unit propagated literal by the newly learned clause. Hence, when a conflicting clause is detected, the backjump clause can be computed exactly as in the propositional case, as explained in Example 2.6.

Unfortunately, this approach, used for instance in the latest version of the MathSAT system [Bozzano et al. 2005], has some important drawbacks. We have done extensive experiments, running our DPLL(T) implementations on all the formulas available in the SMT-LIB benchmark library [Ranise and Tinelli 2003; Tinelli and Ranise 2005] for the logics EUF, RDL, IDL and UFIDL (see the next section). In these experiments, we have counted (i) the number of theory propagation steps and (ii) the number of times theory propagated literals are involved in a conflict, in other words, the number of resolution steps with explanations.

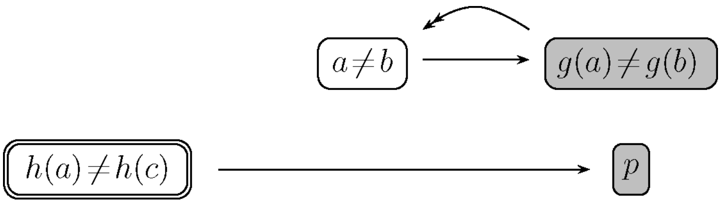
It turns out that, on average, theory propagations are around 250 times more frequent than resolution steps with explanations. For almost all examples, the ratio lies between 20 and 1500. Hence, immediately computing an explanation each time a theory propagation takes place, as done in MathSAT, is bound to be highly inefficient: on average just one of these lemmas out of every 250 is ever going to be used (possibly, even less than that, as each theory propagated literal may occur in more than one conflict). The cost of generating explanations is twofold: it is the cost incurred by $Solver_T$ in computing the clause and that incurred by DPLL(X) in inserting the clause in the clause database and maintaining it under propagation.

There is however a potential advantage in the MathSAT approach, for strategies where TheoryPropagate is applied only if no other rule except Decide is applicable (this is what we did for EUF in Section 4.4). Assume as before that TheoryPropagate applies to a state $M \parallel F$ because $l_1 \wedge \cdots \wedge l_n \models_T l$ for for some subset $\{l_1, \ldots, l_n\}$ of M. Also assume that, due to a previous TheoryPropagate step, the explanation $\neg l_1 \vee \cdots \vee \neg l_n \vee l$ is still present in F, although, due to backtracking, l has become again undefined in M. Then the effect of theory propagating l can now be achieved more efficiently by a unit propagation step with the clause $\neg l_1 \vee \cdots \vee \neg l_n \vee l$. If this leads to a conflict at the current decision level before TheoryPropagate is tried, then a gain in efficiency may be obtained. If, on the other hand, no conflict occurs before applying TheoryPropagate, then it is likely that repeated work is done by $Solver_T$, rediscovering the fact that l is a T-consequence of M.

For some theory solvers, it may be possible that, when computing a T-consequence, there is only a low additional cost in computing its explanation as well at the same time. But even then one usually would not want to pay the time and memory cost of adding the lemma as a new clause—since in many cases this is going to be wasted work and space. One could simply store the lemma as a passive clause, that is, not active in the DPLL procedure, or store some information on how to compute it later.

5.2. Handling Conflict Analysis in the Context of Theory Propagation. In the previous subsection we have argued that it is preferable to generate explanations only at the moment they are needed for conflict analysis. Here we analyze the possible problems that arise in doing so, and discuss when and how it is still possible to compute a backjump clause. In a state of the form $M_1\, l\, M_2\, l'\, M_3 \parallel F$, we say that l is *older* than l', and that l' is *newer* than l.

Too new explanations:
Let us first revisit Example 5.1. After the four steps, where Clause 3 is conflicting, if $Solver_T$ is asked to compute the explanation of $a \neq b$, it can also return $g(a) \neq g(b)$, instead of the "real" explanation $h(a) = h(c) \vee c \neq b \vee a \neq b$. Indeed $a \neq b$ is a T-consequence of $g(a) \neq g(b)$ as well. But $g(a) \neq g(b)$ is a *too new explanation*: it did not even belong to the partial assignment at the time $a \neq b$ was propagated, and was in fact deduced by UnitPropagate from $a \neq b$ itself and Clause 1. Too new explanations are problematic because they can cause cycles in the conflict graph:

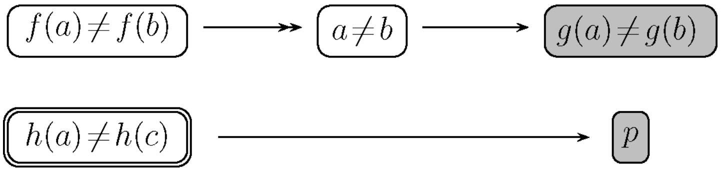


For the conflict graph above, the backwards resolution process computing the backjump clause does in fact loop:

$$\dfrac{g(a)\!=\!g(b) \vee \mathbf{a\!\neq\! b} \qquad \dfrac{a\!=\!b \vee \mathbf{g(a)\!\neq\! g(b)} \qquad \dfrac{h(a)\!=\!h(c) \vee \mathbf{p} \quad g(a)\!=\!g(b) \vee \neg\mathbf{p}}{\mathbf{g(a)\!=\!g(b)} \vee h(a)\!=\!h(c)}}{h(a)\!=\!h(c) \vee \mathbf{a\!=\!b}}}{g(a)\!=\!g(b) \;\vee\; h(a)\!=\!h(c)}$$

Therefore, to make sure that a backjump clause can be found, $Solver_T$ should never return too new explanations. A sufficient condition is to require that all literals in the explanation of a literal l be older than l. In other words, if the current state is of the form $M\, l\, N \parallel F$, then all literals in the explanation of l should occur in M.

Too old explanations:
In our example, when $Solver_T$ was asked to compute the explanation of $a \neq b$, it could also have returned $f(a) \neq f(b)$. This literal was already available before $a \neq b$ was obtained, but, as mentioned in Section 4.4, $Solver_T$ might have failed to detect $a \neq b$ as a negative consequence of it. It is interesting to observe that, with $f(a) = f(b) \vee a \neq b$ as the explanation of $a \neq b$, the resulting conflict graph has no unique implication point (UIP). In fact, there is not even a path from the current decision literal $h(a) \neq h(c)$ to the conflicting literal (of the current decision level) $g(a) \neq g(b)$:



However, looking at what happens with the backwards resolution procedure, one can see that it still produces a backjump clause, that is, a clause with exactly one literal from the current decision level:

$$\dfrac{f(a)\!=\!f(b) \vee \mathbf{a\!\neq\! b} \qquad \dfrac{a\!=\!b \vee \mathbf{g(a)\!\neq\! g(b)} \qquad \dfrac{h(a)\!=\!h(c) \vee \mathbf{p} \quad g(a)\!=\!g(b) \vee \neg\mathbf{p}}{\mathbf{g(a)\!=\!g(b)} \vee h(a)\!=\!h(c)}}{h(a)\!=\!h(c) \vee \mathbf{a\!=\!b}}}{h(a)\!=\!h(c) \;\vee\; f(a)\!=\!f(b)}$$

The following theorem states that in the backwards resolution process too old explanations are never a problem. It follows that just disallowing too new explanations suffices to guarantee that a backjump clause is always found.

THEOREM 5.2. *Assume that for any state of the form $M \parallel F$ and for any l in M due to* TheoryPropagate, *the explanation of l produced by $Solver_T$ contains no literals newer than l in M. Then, if some clause C is conflicting in a state S, either* Fail *applies to S, or else the backwards resolution process applied to C reaches a backjump clause.*

PROOF. Let d be the largest of the decision levels of the literals in C, and let D be the (nonempty) set of all literals of C that have become false at decision level d. If D is a singleton, C itself is a backjump clause. Otherwise, we can apply the backwards conflict resolution process, resolving away literals of decision level d, until we reach a backjump clause having exactly one literal of level d. This process always terminates because each resolution step replaces a literal of decision level d by a finite number (zero in the case of a too old explanation) of strictly older literals of level d. The process is also guaranteed to produce a clause with just one literal of decision level d because, except for the decision literal itself, every literal of decision level d is resolvable. □

Note that the previous theorem is rather general by making no assumptions on the strategy followed in applying the DPLL rules. Also note that the theorem holds in the purely propositional case a well, where the theory T is empty and the theorem's assumption is vacuously true as TheoryPropagate never applies. Its generality entails that, for instance, one can apply Decide even in the presence of a conflicting clause, or if UnitPropagate also applies. In contrast, in Zhang and Malik [2003], the correctness proof of the Chaff algorithm assumes the fixed standard strategy in which unit propagation is done exhaustively before making any new decisions, which is considered an "important invariant". Theorem 5.2 instead shows that it is unproblematic for conflict analysis if a literal l is unit propagated at a decision level d when in fact it could have been propagated already at an earlier level. The reason is simply that in the backwards resolution step resolving on l replaces it by zero literals of level d, in perfect analogy to what happens to theory propagated literals with a too old explanation.

5.3. The Degree of Eagerness by Which Theory Propagation Should be Performed. So far, we have seen two possible strategies for theory propagation.

The first one, which we defined for Difference Logic, requires that $Solver_T$ returns *all* theory consequences (Section 4.2). In that strategy, TheoryPropagate is invoked each time a new literal is added to the current partial assignment. This is done to ensure that the partial assignment never becomes T-inconsistent.

The second strategy, defined for EUF logic, assumes that $Solver_T$ may return only some subset of the theory consequences, and applies TheoryPropagate only if no rule other than TheoryPropagate or Decide is applicable (Section 4.4).

However, there may also be expensive theories where one does not want to do full theory propagation (or check T-consistency) before every Decide step, but instead invoke it in some cheaper, incomplete way. The complete check is only required at the leaves of the search tree, that is, each time a propositional model has been found, in order to decide its T-consistency (this coincides with what is done in the naive lazy approach). The MathSAT approach [Bozzano et al. 2005] is based on a similar hierarchical view, where cheaper checks are performed more eagerly than expensive ones.

6. *Experiments with an Implementation of DPLL(T)*

We have experimented the DPLL(T) architecture with various implementations collaboratively developed in Barcelona and Iowa. We describe here our most advanced implementation, Barcelogic, developed mostly in Barcelona. The system follows the strategies presented in Section 4, and its solvers are as described in Section 4.3 and in Nieuwenhuis and Oliveras [2003, 2005b]. Its DPLL(X) engine implements state-of-the-art techniques such as the two-watched literal scheme for unit propagation, the first-UIP learning scheme, and VSIDS-like decision heuristics

[Moskewicz et al. 2001; Zhang et al. 2001]. The $\mathcal{T}$-Forget rule is currently applied by DPLL(X) after each restart, removing a portion of the learned clauses according to their activity level [Goldberg and Novikov 2002], defined as the number of times they were involved in a conflict since the last restart.

6.1. THE 2005 SMT COMPETITION. The effectiveness of Barcelogic was shown at the 2005 SMT Competition [Barrett et al. 2005]. The competition used problems from the SMT-LIB library [Tinelli and Ranise 2005], a fairly large collection of benchmarks (around 1300) coming from such diverse areas as software and hardware verification, bounded model checking, finite model finding, and scheduling. These benchmarks were in the standard format of SMT-LIB [Ranise and Tinelli 2003], and were classified into 7 competition divisions according to their background theory and some additional syntactic restrictions. For each division, around 50 benchmarks were randomly chosen and given to each entrant system with a time limit of 10 minutes per benchmark.

Barcelogic entered and won all four divisions for which it had a theory solver: EUF, IDL and RDL (respectively, integer and real Difference Logic), and UFIDL (combining EUF and IDL). At least 10 systems participated in each of these divisions. Among the competitors were well-known SMT solvers such as SVC [Barrett et al. 1996], CVC [Barrett et al. 2002], CVC-Lite [Barrett and Berezin 2004], MathSAT [Bozzano et al. 2005], and two very recent successors of ICS [Filliâtre et al. 2001]: Yices (by Leonardo de Moura) and Simplics (by Dutertre and de Moura). Apart from EUF and Difference Logic, these systems also support other theories such as arrays (except MathSAT), and linear arithmetic (SVC only over the reals).

It is well-known that in practical problems over richer (combined) theories usually a large percentage of the work still goes into EUF and Difference Logic. For example, in [Bozzano et al. 2005] it is mentioned that in many calls a general solver is not needed: "very often, the unsatisfiability can be established in less expressive, but much easier, sub-theories". Similarly, Seshia and Bryant [2004], which deals with quantifier-free Presburger arithmetic, states that it has been found by them and others that literals are "mainly" difference logic.

The competition was run on 2.6 GHz, 512 MB, Pentium 4 Linux machines, with a 512 KB cache. For each division, the results of the best three systems are shown in the following table, where **Total time** is the total time in seconds spent by each system, with a timeout of 600 seconds, and **Time solved** is the time spent on the solved problems only:

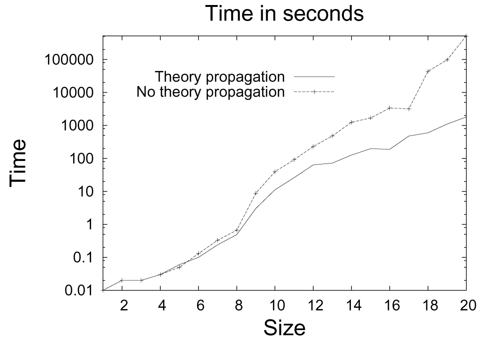
	Top-3 systems	**# problems solved**	**Total time**	**Time solved**
EUF (50 problems):	Barcelogic	39	8358	1758
	Yices	37	9601	1801
	MathSAT	33	12386	2186
RDL (50 pbms.):	Barcelogic	41	6341	940
	Yices	37	9668	1868
	MathSAT	37	10408	2608
IDL (51 pbms.):	Barcelogic	47	3531	1131
	Yices	47	4283	1883
	MathSAT	46	4295	1295
UFIDL (49 pbms.):	Barcelogic	45	2705	305
	Yices	36	9789	1989
	MathSAT	22	17255	1055

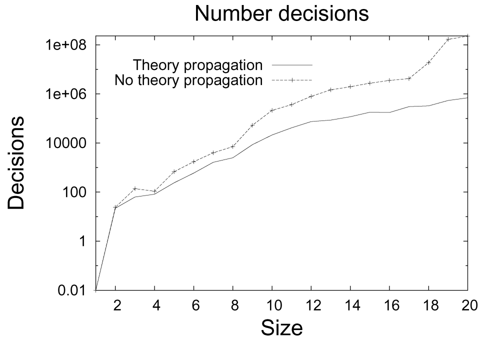
Not only did Barcelogic solve more problems than each of the other systems, it also did so in considerably less time, even—and in spite of the fact that it solved more problems—if only the time spent on the solved problems is counted.

6.2. Experiments on the Impact of Theory Propagation. In our experience, the overhead produced by theory propagation is almost always compensated by a significant reduction of the search space. In Ganzinger et al. [2004] we presented extensive experimental results showing its effectiveness in our DPLL(T) approach for EUF logic. In Nieuwenhuis and Oliveras [2005a] we discussed a large number of experiments for Difference Logic, with additional emphasis on the good scaling properties of the approach. The new SMT solver Yices now also heavily relies on theory propagation.

Of course, theory propagation may not pay off in certain specific problems where the theory plays an insignificant role, that is, where reasoning is done almost entirely at the Boolean level. Such situations can be detected on the fly by computing the percentage of conflicts caused by theory propagations. If this number is very low, theory propagation can be switched off automatically, or applied more lazily, to speed up the computation. (This is done in a forthcoming release of our system.)

In the following two figures, Barcelogic with and without theory propagation is compared, on the same type of machine as in the previous subsection, in terms of run time (in seconds) and number of decisions (applications of Decide) on a typical real-world Difference Logic suite (fisher6-mutex, see Tinelli and Ranise [2005]), consisting of 20 problems of increasing size.





The figures show the typical behavior on the larger problems where the theory plays a significant role: both the run time and the number of decisions are orders of magnitude smaller in the version with theory propagation (note that times and decisions are plotted on a logarithmic scale). In both cases, the DPLL(X) engine used was exactly the same, although in the exhaustive theory case some parts of the code were never executed (e.g., theory lemma learning).

6.3. Experiments Comparing Barcelogic with the Eager Approach. For completeness, we finally compare Barcelogic with UCLID, the best-known tool implementing the eager translation approach to SMT [Lahiri and Seshia 2004]. We show below run time results (in seconds) for three typical series of benchmarks for UFIDL coming from different methods for pipelined processor verification given in [Manolios and Srinivasan 2005a, 2005b] (more precisely, for the BIDW case (i) flushing, (ii) commitment good MA and (iii) commitment GFP). The benchmarks were run on the same type of machine as in the previous two subsections, but this time with a one hour timeout. We used Siege [Ryan 2004] as the final SAT solver for UCLID, since it performed better than any other available SAT solver on these problems.

	UCLID	**BLogic**	**UCLID**	**BLogic**	**UCLID**	**BLogic**
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

We emphasize that these results are typical for the pipelined processor verification problems coming from this source, a finding that has also independently been reproduced by P. Manolios (private communication). We refer the reader to the results given in Ganzinger et al. [2004], showing that our approach also dominates UCLID in the pure EUF case, as well as for EUF with *integer offsets* (interpreted successor and predecessor symbols).

7. *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 to describe several variants of a new, efficient, and modular approach for SMT, called DPLL(T). Given a DPLL(X) engine, a DPLL(T) system for a theory T is obtained by simply plugging in the corresponding theory solver $Solver_T$, which must only be able to deal with *conjunctions* of theory literals and conform to a minimal and simple set of additional requirements.

We are currently working on several—in our opinion very promising—ways to improve and extend both the abstract framework and the DPLL(T) architecture.

The abstract framework can be extended to deal more effectively with theories where the satisfiability of conjunctions of literals is already NP-hard by lifting, from the theory solver to the DPLL(X) engine, some or all of the case analysis done by the theory solver. Along those lines, the framework can also be nicely extended to a Nelson-Oppen style combination framework for handling formulas over several theories. The resulting *DPLL*($T_1, \ldots, T_n$) architecture can deal modularly and efficiently with the combined theories.

Preliminary experiments reveal that other applications of the DPLL(T) framework can produce competitive decision procedures as well for completely different (at least on the surface) kinds of problems. For example, optimization aspects of problems such as pseudo-Boolean constraints can be nicely expressed and efficiently solved in this framework by recasting them as particular SMT problems.

ACKNOWLEDGMENTS. We would like to thank Roberto Sebastiani for a number of insightful discussions and comments on the lazy SMT approach and DPLL(T). We are also grateful to the anonymous referees for their helpful suggestions on improving this article.

REFERENCES

ALUR, R. 1999. Timed automata. In *Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99)* (Trento, Italy), N. Halbwachs and D. Peled, Eds. Lecture Notes in Computer Science, vol. 1633. Springer-Verlag, New York, 8–22.

ARMANDO, A., CASTELLINI, C., AND GIUNCHIGLIA, E. 2000. SAT-based procedures for temporal reasoning. In *Proceedings of the 5th European Conference on Planning* (Durham, UK), S. Biundo and M. Fox, Eds. Lecture Notes in Computer Science, vol. 1809. Springer-Verlag, New York, 97–108.

ARMANDO, A., CASTELLINI, C., GIUNCHIGLIA, E., AND MARATEA, M. 2004. A SAT-based decision procedure for the Boolean combination of difference constraints. In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*. Lecture Notes in Computer Science. Springer-Verlag, New York.

AUDEMARD, G., BERTOLI, P., CIMATTI, A., KORNILOWICZ, A., AND SEBASTIANI, R. 2002. A SAT based approach for solving formulas over Boolean and linear mathematical propositions. In *Proceedings of the CADE-18*. Lecture Notes in Computer Science, vol. 2392. Springer-Verlag, New York, 195–210.

BALL, T., COOK, B., LAHIRI, S. K., AND ZHANG, L. 2004. Zapato: Automatic theorem proving for predicate abstraction refinement. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV'04)* (Boston, MA). R. Alur and D. Peled, Eds. Lecture Notes in Computer Science, vol. 3114. Springer-Verlag, New York. 457–461.

BARRETT, C., DE MOURA, L., AND STUMP, A. 2005. SMT-COMP: Satisfiability modulo theories competition. In *Proceedings of the 17th International Conference on Computer Aided Verification*, K. Etessami and S. Rajamani, Eds. Lecture Notes in Computer Science, vol. 3576. Springer-Verlag, New York, 20–23. (See www.csl.sri.com/users/demoura/smt-comp.)

BARRETT, C., DILL, D., AND STUMP, A. 2002. Checking satisfiability of first-order formulas by incremental translation into SAT. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV)*. Lecture Notes in Computer Science, vol. 2404. Springer-Verlag, New York.

BARRETT, C., DILL, D. L., AND LEVITT, J. 1996. Validity checking for combinations of theories with equality. In *Proceedings of the 1st International Conference on Formal Methods in Computer Aided Design*. Lecture Notes in Computer Science, vol. 1166. Springer-Verlag, New York, 187–201.

BARRETT, C. W. 2003. Checking validity of quantifier-free formulas in combinations of first-order theories. Ph.D. dissertation. Stanford University, Stanford, CA.

BARRETT, C. W., AND BEREZIN, S. 2004. CVC lite: A new implementation of the cooperating validity checker. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV'04)* (Boston, MA). R. Alur and D. Peled, Eds. Lecture Notes in Computer Science, vol. 3114. Springer-Verlag, New York, 515–518.

BAYARDO, R. J. J., AND SCHRAG, R. C. 1997. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI'97)* (Providence, RI), 203–208.

BEAME, P., KAUTZ, H., AND SABHARWAL, A. 2003. On the power of clause learning. In *Proceedings of IJCAI-03, 18th International Joint Conference on Artificial Intelligence* (Acapulco, MX).

BONET, M. L., ESTEBAN, J. L., GALESI, N., AND JOHANNSEN, J. 2000. On the relative complexity of resolution refinements and cutting planes proof systems. *SIAM J. Comput. 30*, 5, 1462–1484.

BOZZANO, M., BRUTTOMESSO, R., CIMATTI, A., JUNTTILA, T. V. ROSSUM, P., SCHULZ, S., AND SEBASTIANI, R. 2005. An incremental and layered procedure for the satisfiability of linear arithmetic logic. In *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference (TACAS)*. Lecture Notes in Computer Science, vol. 3440. Springer-Verlag, New York, 317–333.

BRYANT, R., GERMAN, S., AND VELEV, M. 2001. Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. *ACM Trans. Computational Logic 2*, 1, 93–134.

BRYANT, R., LAHIRI, S., AND SESHIA, S. 2002. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV)*. Lecture Notes in Computer Science, vol. 2404. Springer-Verlag, New York.

BRYANT, R. E., AND VELEV, M. N. 2002. Boolean satisfiability with transitivity constraints. *ACM Trans. Computational Logic 3*, 4, 604–627.

BURCH, J. R., AND DILL, D. L. 1994. Automatic verification of pipelined microprocessor control. In *Proceedings of the 6th International Conference on Computer Aided Verification (CAV)*. Lecture Notes in Computer Science, vol. 818. Springer-Verlag, New York. 68–80.

DAVIS, M., LOGEMANN, G., AND LOVELAND, D. 1962. A machine program for theorem-proving. *Commun. ACM 5*, 7, 394–397.

DAVIS, M., AND PUTNAM, H. 1960. A computing procedure for quantification theory. *J ACM 7*, 201–215.

DE MOURA, L., AND RUEß, H. 2002. Lemmas on demand for satisfiability solvers. In *Proceedings of the 5th International Symposium on the Theory and Applications of Satisfiability Testing (SAT'02)*. 244–251.

DE MOURA, L., AND RUESS, H. 2004. An experimental evaluation of ground decision procedures. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV'04)* (Boston, MA). R. Alur and D. Peled, Eds. Lecture Notes in Computer Science, vol. 3114. Springer-Verlag, New York, 162–174.

DE MOURA, L., RUEß, H., AND SHANKAR, N. 2004. Justifying equality. In *Proceedings of the 2nd Workshop on Pragmatics of Decision Procedures in Automated Reasoning* (Cork, Ireland).

DOWNEY, P. J., SETHI, R., AND TARJAN, R. E. 1980. Variations on the common subexpressions problem. *J. ACM 27*, 4, 758–771.

EÉN, N., AND SÖRENSSON, N. 2003. An extensible SAT-solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT)*. 502–518.

FILLIÂTRE, J.-C., OWRE, S., RUEß, H., AND SHANKAR, N. 2001. ICS: Integrated canonization and solving (tool presentation). In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'2001)*. G. Berry, H. Comon, and A. Finkel, Eds. Lecture Notes in Computer Science, vol. 2102. Springer-Verlag, New York, 246–249.

FLANAGAN, C., JOSHI, R., OU, X., AND SAXE, J. B. 2003. Theorem proving using lazy proof explanation. In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV)*. Lecture Notes in Computer Science, vol. 2725. Springer-Verlag, New York.

GANZINGER, H., HAGEN, G., NIEUWENHUIS, R., OLIVERAS, A., AND TINELLI, C. 2004. DPLL(T): Fast Decision Procedures. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV'04)* (Boston, MA). R. Alur and D. Peled, Eds. Lecture Notes in Computer Science, vol. 3114. Springer-Verlag, New York, 175–188.

GOLDBERG, E., AND NOVIKOV, Y. 2002. BerkMin: A fast and robust SAT-solver. In *Proceedings of the Symposium on Design, Automation, and Test in Europe (DATE '02)*. 142–149.

HODGES, W. 1993. *Model Theory*. Enclyclopedia of mathematics and its applications, vol. 42. Cambridge University Press, Cambridge, MA.

JAFFAR, J., AND MAHER, M. 1994. Constraint Logic Programming: A Survey. *J. Logic Prog. 19/20*, 503–581.

LAHIRI, S. K., AND SESHIA, S. A. 2004. The UCLID Decision Procedure. In *Computer Aided Verification, 16th International Conference (CAV)*. Lecture Notes in Computer Science, vol. 3114. Springer-Verlag, New York, 475–478.

MANOLIOS, P., AND SRINIVASAN, S. K. 2005a. A computationally efficient method based on commitment refinement maps for verifying pipelined machines. In *Proceedings of the ACM IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*. ACM, New York.

MANOLIOS, P., AND SRINIVASAN, S. K. 2005b. Refinement maps for efficient verification of processor models. In *Proceedings of the Design, Automation and Test in Europe Conference and Exposition (DATE)*. IEEE Computer Society, 1304–1309.

MARQUES-SILVA, J., AND SAKALLAH, K. A. 1999. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Comput. 48*, 5 (May), 506–521.

MEIR, O., AND STRICHMAN, O. 2005. Yet another decision procedure for equality logic. In *Proceedings of the 17th International Conference on Computer Aided Verification (CAV'05)* (Edimburgh, Scotland). K. Etessami and S. K. Rajamani, Eds. Lecture Notes in Computer Science, vol. 3576. Springer-Verlag, New York, 307–320.

MOSKEWICZ, M. W., MADIGAN, C. F., ZHAO, Y., ZHANG, L., AND MALIK, S. 2001. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*.

NIEUWENHUIS, R., AND OLIVERAS, A. 2003. Congruence Closure with Integer Offsets. In *Proceedings of the 10h International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, M. Vardi and A. Voronkov, Eds. Lecture Notes in Artificial Intellegence, vol. 2850. Springer-Verlag, New York, 2850. 78–90.

NIEUWENHUIS, R., AND OLIVERAS, A. 2005a. DPLL(T) with exhaustive theory propagation and its application to difference logic. In *Proceedings of the 17th International Conference on Computer Aided Verification (CAV'05)* (Edimburgh, Scotland). K. Etessami and S. K. Rajamani, Eds. Lecture Notes in Computer Science, vol. 3576. Springer-Verlag, New York, 321–334.

NIEUWENHUIS, R., AND OLIVERAS, A. 2005b. Proof-producing congruence closure. In *Proceedings of the 16th International Conference on Term Rewriting and Applications (RTA'05)* (Nara, Japan). J. Giesl, Ed. Lecture Notes in Computer Science, vol. 3467. Springer-Verlag, New York, 453–468.

NIEUWENHUIS, R., OLIVERAS, A., AND TINELLI, C. 2005. Abstract DPLL and abstract DPLL modulo theories. In *Proceedings of the 11th International Conference Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*. F. Baader and A. Voronkov, Eds. Lecture Notes in Computer Science, vol. 3452. Springer-Verlag, New York, 36–50.

PNUELI, A., RODEH, Y., SHTRICHMAN, O., AND SIEGEL, M. 1999. Deciding equality formulas by small domains instantiations. In *Proceedings of the 11th International Conference on Computer Aided Verification (CAV)*. Lecture Notes in Computer Science, vol. 1633. Springer-Verlag, New York. 455–469.

RANISE, S., AND TINELLI, C. 2003. The SMT-LIB format: An initial proposal. In *Proceedings of the 1st Workshop on Pragmatics of Decision Procedures in Automated Reasoning*. Miami.

RYAN, L. 2004. Efficient algorithms for clause-learning SAT solvers. M.S. dissertation, School of Computing Science, Simon Fraser University.

SCHRIJVER, A. 1987. *Theory of Linear and Integer Programming*. Wiley, New York.

SESHIA, S., LAHIRI, S., AND BRYANT, R. 2003. A hybrid SAT-based decision procedure for separation logic with uninterpreted functions. In *Proceedings of the 40th Design Automation Conference (DAC)*. 425–430.

SESHIA, S. A. 2005. Adaptive eager Boolean encoding for arithmetic reasoning in verification. Ph.D. dissertation, Carnegie-Mellon University, Pittsburgh, PA.

SESHIA, S. A., AND BRYANT, R. E. 2004. Deciding quantifier-free presburger formulas using parameterized solution bounds. In *Proceedings of the 19th IEEE Symposium on Logic in Computer Science (LICS 2004)*. IEEE Computer Society Press, Los Alamitos, CA, 100–109.

STRICHMAN, O. 2002. On solving presburger and linear arithmetic with SAT. In *Proceedings of the Formal Methods in Computer-Aided Design, 4th International Conference (FMCAD 2002)* (Portland, OR). M. Aagaard and J. W. O'Leary, Eds. Lecture Notes in Computer Science, vol. 2517. Springer-Verlag, New York, 160–170.

STRICHMAN, O., SESHIA, S. A., AND BRYANT, R. E. 2002. Deciding separation formulas with SAT. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV)*. Lecture Notes in Computer Science, vol 2404. Springer-Verlag, New York. 209–222.

STUMP, A., AND TAN, L.-Y. 2005. The algebra of equality proofs. In *Proceedings of the 16th International Conference on Term Rewriting and Applications, RTA'05* (Nara, Japan). J. Giesl, Ed. Lecture Notes in Computer Science, vol. 3467. Springer-Verlag, New York, 469–483.

TALUPUR, M., SINHA, N., STRICHMAN, O., AND PNUELI, A. 2004. Range allocation for separation logic. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV 2004)*. (Boston, MA July 13–17). Lecture Notes in Computer Science, Springer-Verlag, New York, 148–161.

TINELLI, C. 2002. A DPLL-based calculus for ground satisfiability modulo theories. In *Proceedings of the 8th European Conference on Logics in Artificial Intelligence*. Lecture Notes in Artificial Intelligence, vol. 2424. Springer-Verlag, New York, 308–319.

TINELLI, C., AND RANISE, S. 2005. *SMT-LIB: The Satisfiability Modulo Theories Library*. http://goedel.cs.uiowa.edu/smtlib/.

ZHANG, H. 1997. SATO: An efficient propositional prover. In *Proceedings of the 14th International Conference on Automated Deduction*. Springer-Verlag, New York, 272–275.

ZHANG, L., MADIGAN, C. F., MOSKEWICZ, M. W., AND MALIK, S. 2001. Efficient conflict driven learning in a Boolean satisfiability solver. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'01)*. 279–285.

ZHANG, L., AND MALIK, S. 2003. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Proceedings of the 2003 Design, Automation and Test in Europe Conference (DATE 2003)*. IEEE Computer Society Press, Los Alamitos, CA, 10880–10885.

RECEIVED DECEMBER 2005; REVISED FEBRUARY 2006 AND JULY 2006; ACCEPTED NOVEMBER 2006